Provable Protection of Confidential Data in Microkernel-Based Systems

Diplom Informatiker
Marcus Rolf Völp

Dissertation

submitted to the
Faculty for Computer Science
Technische Universität Dresden

in partial fulfillment of the requirements for the degree of:
Doktoringenieur (Dr.-Ing.)

August, 10, 2010

Thesis Committee:

Prof. Dr. rer. nat. Hermann Härtig	Technische Universität Dresden
Prof. Dr. rer. nat. habil. Christel Baier	Technische Universität Dresden
Prof. Dr.-Ing. Christian Hochberger	Technische Universität Dresden
Prof. Dr.-Ing. habil. Klaus Kabitzsch	Technische Universität Dresden
Prof. Dr. ir. Erik Poll	Radboud Universiteit Nijmegen

Abstract

Although modern computer systems process increasing amounts of sensitive, private, and valuable information, most of today's operating systems (OSs) fail to protect confidential data against unauthorized disclosure over covert channels. Securing the large code bases of these OSs and checking the secured code for the absence of covert channels would come at enormous costs. Microkernels significantly reduce the necessarily trusted code. However, cost-efficient, provable confidential-data protection in microkernel-based systems is still challenging.

This thesis makes two central contributions to the provable protection of confidential data against disclosure over covert channels:

- A budget-enforcing, fixed-priority scheduler that provably eliminates covert timing channels in open microkernel-based systems; and

- A sound control-flow-sensitive security type system for low-level operating-system code.

To prevent scheduling-related timing channels, the proposed scheduler treats possibly leaking, blocked threads as if they were runnable. When it selects such a thread, it runs a higher classified budget consumer.

A characterization of budget-consumer time as a blocking term makes it possible to reuse a large class of existing admission tests to determine whether the proposed scheduler can meet the real-time guarantees of all threads we envisage to run. Compared to contemporary information-flow-secure schedulers, significantly more real-time threads can be admitted for the proposed scheduler.

The role of the proposed security type system is to prove those system components free of security policy violating information flows that simultaneously operate on behalf of differently classified clients. In an open microkernel-based system, these are the microkernel and the necessarily trusted multilevel servers.

To reduce the complexity of the security type system, C++ operating-system code is translated into a corresponding *Toy* program, which in turn is complemented with calls to *Toy* procedures describing the side effects of interactions with the underlying hardware. *Toy* is a non-deterministic intermediate programming language, which I have designed specifically for this purpose. A universal lattice for shared-memory programs enables the type system to check the resulting *Toy* code for potentially harmful information flows, even if the security policy of the system is not known at the time of the analysis.

I demonstrate the feasibility of the proposed analysis in three case studies: a virtual-memory access, L4 inter-process communication and a secure buffer cache. In addition, I prove Osvik's countermeasure effective against AES cache side-channel attacks. To my best knowledge, this is the first security-type-system-based proof of such a countermeasure. The ability of a security type system to tolerate temporary breaches of confidentiality in lock-protected shared-memory regions turned out to be fundamental for this proof.

Acknowledgements

First and foremost, I would like to thank my kids, my wife, and my parents for their continuing love and support.

I would like to thank my advisor, Prof. Hermann Härtig, for his support and gainful criticism. Furthermore, I would like to thank the second reviewer of this thesis, Prof. Erik Poll, for his detailed and helpful comments and for the opportunity to learn many things during my visits of the Digital Security Group in Nijmegen.

This thesis would not have become reality without the help of many people. Special thanks go to the members of the Operating Systems Group at TU Dresden. In particular, I would like to thank Dr. Claude-Joachim Hamann for the many insightful discussions about schedulers and scheduling theory and Benjamin Engel for his help in moving this work into the right direction. Thanks also for your helpful comments when we switched sides and you started commenting on my thesis.

Last but not least, I want to thank Hendrik Tews and Tjark Weber for the joy and pleasure I had working with you in Robin. Thanks Hendrik for prove reading my work and especially for teaching an operating-systems guy like me this strange topic called formal methods.

Finally, those of you who like me had the pleasure of being invited to one of Hermann's famous parties will surely agree with me and with the dolphins leaving earth [Ada79]: "So long, and thanks for all the fish".

Erklärung

Hiermit versichere ich, dass die hier präsentierte Arbeit das Ergebnis meiner alleinigen und orginären Forschung ist. Diese Arbeit ist ohne die Zuhilfenahme unzulässiger Hilfe endstanden. Alle genutzen Hilfsmittel sind als solche gekennzeichnet. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind in meiner Arbeit als solche kenntlich gemacht.

Ich versichere weiterhin, dass ich die vorliegende Arbeit nicht in gleicher oder in ähnlicher Form einer anderen Prüfungsbehörde zum Zwecke der Promotion vorgelegt habe. Diese Arbeit wurde noch nicht veröffentlicht. Sie ist Bestandteil meines ersten Promotionsversuchs.

Marcus Völp

Contents

1. Introduction **1**
1.1. Microkernels . . . 2
1.2. Security Type Systems . . . 4
1.3. Challenges and Contributions . . . 6
1.3.1. A Secure Budget-Enforcing Fixed-Priority Scheduler . . . 6
1.3.2. A Sound Security Type System for Low-Level Operating-System Code 8
1.4. Scope and Limitations . . . 10
1.4.1. Perfect Information-Flow Security . . . 11
1.4.2. L4-Family Microkernels . . . 12
1.4.3. Security Type Systems and Static Analyses . . . 12
1.4.4. Assumptions . . . 13
1.5. Synopsis . . . 13

2. Foundations and Related Work **15**
2.1. Covert Channels . . . 15
2.1.1. Storage and Timing Channels . . . 15
2.1.2. Noise . . . 16
2.1.3. Hardware- and Software-Centric Covert Channels . . . 16
2.1.4. Illegal Information Flows in Source Code . . . 16
2.2. Information-Flow Policies . . . 18
2.2.1. Lattice and Non-lattice Models . . . 19
2.2.2. Intransitive Information-Flow Policies . . . 19
2.2.3. Downgrading and Dynamic Policies . . . 19
2.3. Non-interference . . . 21
2.3.1. Non-influence . . . 21
2.3.2. Cryptography and Non-interference . . . 24
2.3.3. Unwinding . . . 25
2.4. Security Type Systems and Related Static Information-Flow Analyses . . . 26
2.4.1. Control-Flow-Insensitive Security Type Systems . . . 27
2.4.2. Control-Flow-Sensitive Security Type Systems . . . 28
2.4.3. Related Information-Flow Analyses . . . 29
2.4.4. A-Priori Unknown Information-Flow Policies . . . 30
2.4.5. Operating-System Functionality . . . 31
2.4.6. Timing-Leak Transformations . . . 32
2.4.7. Points-To Analysis . . . 33
2.4.8. Loop-Bound Analysis . . . 34
2.5. L4-Family Microkernels . . . 34
2.5.1. A Typical L4 Server . . . 36
2.5.2. Confinement . . . 39
2.6. Non-interference-Secure Scheduling . . . 40
2.7. Prototype Verification System (PVS) . . . 41

3. Avoiding External Timing Channels in Fixed-Priority Schedulers **45**
3.1. The *ReThMo* Task Model . . . 46
3.1.1. Task Models for Non-interference Proofs . . . 46
3.1.2. Thread Scheduling Parameters . . . 48
3.1.3. Expressiveness . . . 52
3.2. External Timing Channels in Fixed-Priority Schedulers . . . 57
3.2.1. Indirect Influence . . . 57
3.2.2. Influence due to Non-preemptive Execution . . . 59
3.3. A Non-Interference-Secure Scheduler . . . 60
3.3.1. Avoiding Information Leakage due to Direct and Indirect Influences . . . 61
3.3.2. Suitable Predicates for *Countermeasure I* . . . 63
3.3.3. Transitive Information-Flow Policies . . . 64
3.3.4. Intransitive Information-Flow Policies . . . 65
3.3.5. Avoiding Information Leakage due to Non-preemptive Execution . . . 69
3.3.6. Accounting . . . 74
3.3.7. A Budget-Enforcing Fixed-Priority Lattice Scheduler . . . 74
3.3.8. Limited Number of Priorities . . . 75
3.3.9. Internal-Timing Channels . . . 78
3.3.10. Information-Flow Secure Proportional-Share Schedulers . . . 78
3.4. A Machine-Checked Proof of Non-interference . . . 79
3.4.1. Overview . . . 80
3.4.2. State . . . 82
3.4.3. State Transformers . . . 85
3.4.4. Invariants . . . 92
3.4.5. Non-interference . . . 93
3.4.6. Proof of Non-interference . . . 95
3.4.7. Temporal Isolation of Non-interfering Threads . . . 98
3.5. Real-Time Guarantees . . . 98
3.5.1. Time-Demand Analysis and Liu-Layland Criterion . . . 98
3.5.2. Prohibition Times . . . 99
3.5.3. Blocking due to Self Suspension . . . 100
3.5.4. Blocking due to Non-preemptive Execution . . . 102
3.6. Practical Matters . . . 102
3.6.1. Precedence Constraints . . . 102
3.6.2. Dynamic Thread Creation . . . 103
3.6.3. Hierarchical Scheduling of Differently Classified Threads . . . 104
3.6.4. Timeslice Donation . . . 105
3.7. Resources . . . 107
3.7.1. Self Suspension . . . 107
3.7.2. Priority-Inheritance Protocol . . . 108
3.7.3. Stack-Based Priority Ceiling Protocol . . . 109
3.7.4. Basic Priority Ceiling Protocol and Donation Ceiling . . . 109
3.8. Summary . . . 115

4. Statically Checking Confidentiality of Low-Level Operating-System Code **117**
4.1. A Running Example . . . 118
4.2. Peculiarities of Low-Level Operating-System Code . . . 119
4.2.1. Interactions with the Underlying Kernel and with other Programs . . . 119

4.2.2. Interactions with the Underlying Hardware 121
4.2.3. Low-Level Language Features in Operating-System Code 124
4.2.4. Incomplete Knowledge about the Information-Flow Policy 128
4.2.5. A Protection-Parametric Information-Flow Analysis 129
4.3. Typing a Size-Aligned Virtual-Memory Read 131
4.4. Assumptions 133
4.5. Syntax and Semantics of Toy 134
4.5.1. Memory Model 135
4.5.2. Data Types 137
4.5.3. Dynamic Semantics 138
4.5.4. Shared Memory 143
4.5.5. C++ to *Toy* 148
4.6. Learned Secrets 151
4.6.1. Secrets of the Initial State 151
4.6.2. Evolution of Learned Secrets 152
4.6.3. Constraining the Input Oracle to Produce l-Similar Inputs 153
4.6.4. Example 155
4.7. Security Type System for *Toy* 156
4.7.1. Control-Flow Non-Determinism 156
4.7.2. Typing Rules for the Deterministic Core of *Toy* 158
4.7.3. Soundness 163
4.8. Summary 168

5. Case Studies **169**
5.1. Page-Table Walk 169
5.2. IPC 172
5.3. Buffer Cache 178
5.4. AES 180

6. Conclusions and Future Work **185**

A. Avoiding the Deactivation of Nonpreemptively Executing Threads **189**

List of Figures

2.1. Control-Flow Insensitive Security Type System 27
2.2. Flow-Sensitive Security Type System 28
2.3. Template of an L4 Server 36
2.4. Server Loop 38

3.1. Thread State Diagram 49
3.2. Scheduling Parameters 50
3.3. Enforcing Blocking Limits 51
3.4. Indirect Influence Scenario 58
3.5. Exceeded Execution Budget and Deadline by Delaying Preemptions 59
3.6. Avoiding Information Leakage due to Direct and Indirect Influences 62
3.7. Thread Activation 63
3.8. Mikro-SINA 65
3.9. Cryptographic Gateway 67
3.10. Countermeasure to Avoid Leakage due to Non-Preemptive Execution 70
3.11. Blocking due to Self Suspension 100
3.12. Self Suspension on Unavailable Resources 108
3.13. Priority-Ceiling Protocol 110
3.14. Donation Ceiling 112

4.1. Propagation of Accessed Bits 132
4.2. Small Step Semantics of *Toy* Expressions. 139
4.3. Small Step Semantics of *Toy* Statements 141
4.4. A simple C++ pointer program and its *Toy* translation. 149
4.5. Stepwise-interleaved evaluation of L^i, M^i, s^i and t^i. 154
4.6. Typing Rules for *Toy* Expressions 160
4.7. Typing Rules for the deterministic *Toy* Statements 161

5.1. *Toy* program of the IA32 page-table walk hardware side effect. 171
5.2. Kernel entry and exit path for system calls of the Nova Microhypervisor 173
5.3. Source code of the IPC call operation of the Nova Microhypervisor 175
5.4. Buffer Cache 178
5.5. A hardware side effect for cache eviction 182
5.6. AES encryption 183

1. Introduction

Today's mobile, desktop, and server systems are widely used to process data of high personal, commercial, or industrial value. Bank credentials, private email, content protected audio and video files, health care, and financial data are only a few examples of data whose confidentiality is worth protecting. Yet, despite many years of research on identifying [Kem83, KT96, KP91, TGC87, GM82], analyzing [Tro93, AB03, Mil89a], and mitigating [Hu91, Gra93, PN92] covert channels [Lam73], and despite an equally long history of academic and industrial efforts to build small, secure, and reliable operating-system kernels[1] [SCS77, Fra83, Inc95, Kar88, Har85, FN79, SGLS77, KZB+91, SVJ+05, Inc09, LEA07], covert channels remain a serious security concern.

A *covert channel* is a communication channel that allows threads to transfer information in a manner that violates the system's security policy [TGC87, Gal93]. In the presence of potentially harmful covert channels, no guarantees can be given as to whether attackers may learn information about the sensitive data a system processes, or, in other words, whether the confidentiality of sensitive data is preserved. On the other hand, a system can be secure even though covert channels exists. Covert channels are benign if they cannot be utilized or if the security policy has already authorized information flows between the communicating threads. To provably protect confidential data against leakage, we must therefore either demonstrate the absence of utilizable covert channels, or we must show that no thread with legitimate access to confidential data transfers information about this data over such a channel. Threads that do transfer information over a covert channel are said to *leak* this information.

Various covert channels have been identified in modern computer systems. In Section 2.1, I elaborate on the nature of these channels in greater detail. For now, let us only distinguish *software-centric covert channels* (such as locks on shared files, unintentionally shared regions of memory, or, more generally, software-implemented resources that reveal how other threads use them) from *hardware-centric covert channels* (such as disk-arm movement [KC91], electromagnetic radiation [Age72], or power consumption [KJJ99]).

There are two outstanding reasons why covert channels and illegal information flows remain an issue in today's systems: the high costs of traditional formal and semi-formal methods to assure the absence of potentially harmful covert channels; and, the size and complexity of operating systems (OSs) in modern computer systems. Covert channel analysis costs are significant, both in terms of highly skilled personnel and in terms of labor hours, even if the analysis is carried out only on relatively small amounts of code [Smi01, HKMY87]. Yet, most of today's computer systems run large and complex legacy OSs. The kernel of these OSs often exceeds 200,000 lines of code (LOC) [SPHH06] and contains presumably between 400 and 1200 bugs [CYC+01]. It is therefore little surprising that even security-enhanced legacy OSs [LS01] fail to protect confidential data against covert channels [GHRS05] and that only a small fraction of today's OSs address covert channels at all [KS02].

[1] The kernel of an operating system is the code that runs in the most privileged processor mode.

In this dissertation, I strive for the provably perfect protection of confidential data against software-centric covert channels in low-level operating-system code. Perfect means that even the leakage of a single bit of information is considered harmful.

To provably protect confidential data in operating systems, I propose to combine the complementary strength of two technologies: microkernels and security type systems, a static language-based information-flow analysis. Hence, this thesis is about provable confidential-data protection in microkernel-based systems.

In this combined approach, the role of the microkernel is to avoid covert channels by isolating differently classified applications, legacy OS instances and operating-system servers[2]. The role of security type systems is to prove the absence of those security policy violating information flows that isolation cannot sensibly avoid. These are the illegal information flows that follow from invocations of the microkernel or from invocations of operating-system servers that simultaneously operate on behalf of differently classified clients and that cannot be reinstantiated for each such client. In the following, I shall call these servers the *multilevel servers* of the analyzed microkernel-based system. The microkernel and the multilevel servers I shall call collectively the *multilevel components* of such a system.

The remainder of this introduction is organized as follows: next, I give a more detailed introduction on microkernels, on security type systems, and on the roles they play in the provable protection of confidential data in microkernel-based systems. Section 1.3 summarizes the contributions that this thesis makes and highlights the challenges that must be addressed. Section 1.4 discusses the scope of this thesis and the limitations of the results it presents, Section 1.5 concludes this introduction by giving an outline of the remainder of this thesis.

1.1. Microkernels

The design philosophy of microkernel-based systems [WCC+74] is to implement a universally applicable and absolutely reliable kernel — the microkernel. This kernel should implement only those mechanisms that allow for a convenient, flexible, and efficient implementation of OS facilities and policies outside the kernel. The determining criteria for tolerating a mechanism in the kernel is whether a required system functionality cannot be implemented if this mechanism would reside outside the kernel [Lie95].

Although first-generation microkernels [ABB+86, ZPS99, BCE+94, SESS96, ADH89] were rather large, inflexible, and slow, second-generation microkernels [Lie95, Hil92, HK93, Sha99, KV05, DdEE, SK08, PSLW09, Han99] have been able to demonstrate that these characteristics are not inherent. Second-generation microkernels achieve their goals with only three abstractions and two mechanisms:

- *Address Spaces:* mappings of address-space local identifiers to resources;
- *Threads:* activities that execute inside address spaces; and
- *Kernel Memory:* memory that the kernel can use to create threads, address spaces, user memory and other kernel-implemented objects.

[2]Servers are application-level programs, which provide some OS functionality to other application-level programs.

Inter-process communication (IPC) and an *access-control mechanism* for kernel objects are the two mechanisms, which complement these abstractions. IPC implements a controlled exchange of messages between threads executing in different address spaces. The in-kernel access-control mechanism enforces the part of the security policy that seeks to control which operations threads can execute on a kernel-implemented object. Thereby, the unit of protection enforcement is the address space (i.e., all threads of an address space can exercise the same privileges). Examples of access-control mechanisms are access-control lists, capabilities [DH66, Sha99, SA07, DdEE, KV05, LW09, WL10, Ste09a], reference monitors [Lie92, SVJ+05] and access controls based on static [Inc95] or dynamic secrecy levels [VEK+07, ZBWKM06].

The size of second-generation microkernels is in the order of 14,000 LOC [SPHH06]. This is about a quarter the size of the kernel of the Vax VMM operating system [KZB+91]. Second-generation microkernels host a variety of systems [Sha99, HBB+98, HHF+05, Hil92, HERH93]. And, even paravirtualized [LUY+08, Hoh96, Lac04] and unmodified legacy OSs [PSLW09, SK08] run on top of microkernels or microhypervisors. A *microhypervisor* is a microkernel that supports unmodified guest OSs and deprivileged virtual-machine monitors [SGLS77].

A particularly interesting (and from an information-flow perspective also very challenging) class of microkernel-based systems are open systems as described in Deng et al. [DL97] and in Härtig et al. [HHF+05]. Open systems co-host not necessarily trustworthy legacy OSs and their applications next to security-sensitive and real-time-critical applications on top of a microkernel. As a consequence, microkernels for open systems must not only encapsulate potentially untrustworthy legacy OS instances; they must also meet the timing requirements of simultaneously executing real-time applications.

The co-hosting ability of open systems facilitates a construction principle, which significantly reduces the trusted computing base of security-sensitive or real-time-critical application scenarios: to split sensitive applications into critical and into non-critical parts and to reuse potentially untrustworthy legacy code for the non-critical parts [HBB+98, HPHS04]. In these split-application scenarios, it is customary to cryptographically[3] protect [WH08, SPHH06] confidential data before the potentially untrustworthy legacy code can access it. However, in some scenarios, it is also feasible to grant potentially untrustworthy applications and legacy OS instances plaintext access to confidential data [HWS03]. Then, the primary responsibility of the microkernel and of the multilevel servers is to isolate the parts of split applications in such a way that confidential data cannot be leaked.

To avoid leakage, applications must be isolated both in a temporal and in a spacial manner. The enforcement of temporal isolation is the responsibility of the kernel-level scheduler. In real-time systems, the term temporal isolation is merely used to express the requirement that threads cannot violate the real-time guarantees (e.g., completion within a specified amount of time) of unrelated threads. However, as we shall see in Section 3.2 on page 57, the protection of confidential data against leakage requires a stronger form of temporal isolation: timing must not be a covert channel.

To isolate applications or parts of application-level programs in a spatial manner, all accesses to kernel objects, server-implemented resources, and to other application-level programs must have been authorized by the system's security policy. To enforce this isolation with the in-kernel access-control mechanism, the to-be-isolated parts must be run in separate address spaces and local identifiers must refer only to legitimately accessible objects. This way, leakage is limited

[3] See Section 2.3.2 for a discussion about the relation between cryptography and perfect information-flow security.

to those objects to which the isolated threads in an address space have direct or indirect access. However, because in-kernel access-control mechanisms can only enforce restrictions on the release of information through system calls, information flows between legitimately accessible objects are beyond the control of these mechanisms [DD77]. Hence, in-kernel access-control mechanisms cannot prevent the microkernel from leaking information from one kernel object to another, nor can they prevent multilevel servers from leaking client information into the objects that the server implements for a differently classified client. This is where security type systems come into play.

1.2. Security Type Systems

Inspired by the early work of the Dennings [DD77], security type systems [VSI96] and related language-based approaches to information-flow security have evolved into powerful tools to statically check applications for the absence of security policy violating information flows. For an excellent overview see Sabelfeld and Myers [SM03].

Essentially, security type systems work in the same way as the data type systems of modern compilers: maintaining only the types of variables, both, security type systems and data type systems, abstract from concrete values and from the concrete expressions that compute these values; both infer the types of expression results from the types of the expression parameters; and, both check whether the types of these results are compatible with the types of the variables in which these results are stored.

The fundamental differences between security type systems and data type systems are the types on which they operate: data type systems operate on the common language data types **int**, **float**, **bool**, etc.; security type systems, on the other hand, operate on the secrecy levels of stored information respectively on the secrecy levels up to which eventual observers of a variable are cleared.

Security type systems infer the secrecy level of an expression result as the least upper bound of the secrecy levels of the expression parameters. Hence, they pessimistically assume that the expression produces an encoding, which reveals information about any data in these parameters. To also prevent leakages through the control flow of a program, security type systems also check variable assignments for implicit information flows. Whenever information is assigned to an observable variable, security type systems validate that the assignment happens in a context whose secrecy level is also legitimately observable. The *context* denotes where in the program the assignment is located. Its secrecy level is the least upper bound of the secrecy levels of the conditionals (e.g., of if-statements) that have directed the control flow of the program to this context. Hence, if legitimately observable data is written in a context that depends on a secret conditional, the secrecy level of this conditional is checked together with the secrecy level of the stored information by checking the least upper bound of both secrecy levels against the clearance of eventual observers. A secrecy level of a result is compatible with the clearance of eventual observers of a variable if all observer clearances dominate the secrecy level of this result. This is precisely the case if the greatest lower bound of observer clearances dominates the result secrecy level. I will therefore call this lower bound the *clearance* of this variable.

The *lattice model* [Den76] ensures that least upper bounds and greatest lower bounds always exist. A set of secrecy levels S and the partial order *dominates* $\leq$ form a lattice $(S, \leq)$ if and only if any non-empty finite subset $S' \subseteq S$ of secrecy levels has a unique least upper and greatest lower bound.

Sound security type systems accept only those programs that are free of security policy violating information flows. However, security type systems typically ignore the timing behavior of programs, and hence also the information programs leak through their timing behavior [4]. As a consequence, security type systems are typically only sound with regards to a timing-insensitive (and often also termination-insensitive) information-flow property: timing and termination-insensitive non-interference [GM82]. Non-interference attests the complete absence of security policy violating information flows by requiring the checked program to produce the same output as seen by an arbitrary l-classified observer despite variations in $\not\leq l$ classified inputs.

To also address security policy violating information flows through the program's timing behavior, Agat [Aga00a] suggests a class of program transformations for timing-insensitive non-interference-secure programs called *timing-leak transformations*. Provided a timing-insensitive security type system has already proven a program to be timing-insensitive non-interference secure, a timing-leak transformation eliminates the illegal information flows that encode secrets in the timing of internal and external events. Essentially, such a transformation replaces all secrecy-dependent operations of the to-be-transformed program with semantically equivalent operations that exhibit a secrecy-independent timing behavior. As a result, the timing of observable side effects of these operations can no longer depend on the timing of preceding secrecy-dependent operations. The transformed program is timing-sensitive non-interference secure.

Still, security type systems have their limitations, which justify their combined application with in-kernel access-control mechanisms:

Completeness Security type systems are not *complete*, that is, they cannot classify all information-flow secure programs as secure.

For example, typical security type systems will reject the two secure programs [5] **l = h; l = l − h** and **l = h; l = 0**, although both evaluate to **l == 0** irrespective of the secret value in **h**. Typical security type systems reject the first because they abstract from the concrete values in **l** and **h** and from concrete arithmetic operations **+** and −. Therefore, they cannot detect that the subtraction removes the secret value **h** from **l**. Control-flow-insensitive security type systems cannot accept the second example because they require all subprograms of a checked program to be secure on their own. Obviously, **l = h** is not secure if the temporary breach of confidentiality is not repaired in a subsequent assignment.

Size and Complexity Contemporary security type systems fail to accept some programs just because they are too large or to complex. In the foreseeable future, legacy OSs will likely remain in this class of uncheckable programs, even if one would undertake the challenge to secure them. However, the possibility to reuse these legacy OSs in open microkernel-based systems demonstrates the value of a suitable isolation mechanism besides program analysis.

Unsafe Compiler Optimizations Aggressive and thus potentially unsafe compiler optimizations can jeopardize the confidentiality guarantees of successfully-checked programs. However, in our setting, a restriction to safe compiler optimizations is justified

[4] The security type system in Hedin et al. [HS05] is an exception.

[5] In these programs and in similar examples in the remainder of this thesis, **l** and **h** are two variables with *low* respectively *high* secrecy levels. The security policy authorizes information flows from *low* to *high* but not vice versa.

only for the microkernel and for the multilevel servers: If a server can be re-instantiated for differently classified clients, a single instance of this server needs to hold only those information to which the clients of this instance are cleared anyway. Hence, if we assume that the kernel-level scheduler prevents scheduling-related covert channels, and if we further assume that neither the microkernel nor the multilevel servers can be used by their clients to illegally pass confidential information to other clients, an access-control mechanism, which allows clients to access only their respective server instances, suffices to prevent leakage. A central contribution of this thesis is to construct a static information-flow analysis, which establishes the second assumption for the microkernel and for the multilevel servers. However, the above reasons show that other programs need not to be subjected to such an analysis. Hence, they do not depend on safe compiler optimizations to preserve their confidentiality guarantees.

Low-Level OS Code Finally, as we shall see in greater detail in Section 4.2 on page 119, today's security type systems cannot immediately be applied to the low-level operating-system servers of a microkernel-based system, nor can they produce sound results for the microkernel itself.

Taken together, a successfully checked microkernel with a temporally isolating scheduler and a sound security type system for low level operating-system code compensate the limitations of the respective other technology to provably protect confidential data in open microkernel-based systems.

1.3. Challenges and Contributions

This dissertation makes two central contributions:

1. A modified budget-enforcing fixed-priority scheduler that provably eliminates scheduling-related covert timing channels in open microkernel-based systems; and

2. A sound, control-flow-sensitive security type system to check low-level operating-system code for security policy violating information flows.

In the following, I give an extended introduction to these contributions.

1.3.1. A Secure Budget-Enforcing Fixed-Priority Scheduler

The abilities and limitations of access-control mechanisms to prevent illegal information flows that do not exploit timing behavior are well understood [Den76, Rus92, VEK+07, ZBWKM06]. However, timing leaks are beyond the control of these mechanisms. The first central contribution of this thesis is therefore a budget-enforcing fixed-priority scheduler that provably eliminates scheduling-related timing leaks in open microkernel-based systems. A scheduler is *budget enforcing* if it prevents threads with exhausted execution budget from running.

Operating-systems typically take one of the following two approaches to avoid scheduling-related covert channels: they add noise to all clocks and to all other timing sources [Hu91], or they partition the system in both a spatial and in a temporal manner [Gal93]. Security type systems for programs that run on specific classes of schedulers [SV98, SS00, RS06] and security type systems for programs that run on arbitrary schedulers [SS00] complement these OS-level solutions.

However, because open systems also run real-time-critical applications, neither the two OS-level solutions nor the language-based approaches are perfectly suited for open microkernel-based systems:

1. Fuzzy time [Hu91] reduces the bandwidth of scheduling-related covert channels at the cost of precise timing. Real-time workloads, which require exact timing information to take time stamps of incoming events and to trigger external signals at precise points in time, are thereby jeopardized [BCG+94].

   Moreover, fuzzy time alone cannot effectively mitigate scheduling-related covert channels. Trostle [Tro93] substantiates this point in his model of fuzzy time systems. He shows that a high-bandwidth covert channel (with data rates in the order of 50 bits per second) remains even if clock fluctuations are high (e.g., randomly distributed between 1 ms and 19 ms).

2. Time-partitioned systems [Kop98] temporally isolate threads in different partitions without affecting clock precision. Hence, by assigning differently classified threads to different partitions, time-partitioned systems avoid scheduling-related covert timing channels. However, time-partitioning schedulers cannot run differently classified threads during those times when all threads of the active partition block. A scheduler that can reap benefit of these blocking times can therefore guarantee the in-time completion of significantly more real-time threads. The proposed fixed-priority scheduler reaps benefit of these blocking times.

3. Only successfully checked programs can safely be run if security type systems are the only means to avoid scheduling-related covert channels.

This thesis proposes two modifications to enable a budget-enforcing fixed-priority scheduler to provably eliminate scheduling-related covert channels:

Countermeasure 1: The first modification causes the scheduler to treat possibly leaking blocked higher prioritized threads as is they were runnable.

Countermeasure 2: The second modification causes the scheduler to defer the points in time when higher prioritized threads resume their execution.

As a result of the first countermeasure, other threads in the system can no longer distinguish whether a threads did actually run or whether the scheduler has merely treated this thread as if it were runnable. Consequently, alterations in a thread's scheduling behavior no longer constitute a covert channel. In situations where a non-preemptively-executing low-prioritized thread attempts to leak information by delaying the resumption of a blocked higher prioritized thread, the second countermeasure prevents this leakage by always delaying this resumption to a safe point in time.

As we shall see in greater detail in Chapter 3, a budget-enforcing fixed-priority scheduler that implements these two countermeasures prevents all scheduling-related timing channels. Thereby, it preserves precise timing and most of the real-time guarantees an unmodified fixed-priority scheduler can give. Moreover, because the effect of the first countermeasure on lower prioritized threads can be quantified as a blocking term, a large class of existing admission tests can be reused. An *admission test* determines a-priori whether a scheduler will meet the real-time guarantees of all threads that this scheduler should run.

In the area of information-flow secure schedulers, the detailed contributions of this thesis are:

- *ReThMo*, a task model to describe real-time workloads for the purpose of proving budget-enforcing fixed-priority schedulers non-interference secure;
- *An analysis of scheduling-related covert channels in fixed-priority schedulers*;
- *A non-interference-secure budget-enforcing fixed-priority scheduler*;
- *A formal model of this scheduler and a corresponding machine-checked non-interference proof*;
- *An analysis of the real-time guarantees that this scheduler achieves*;
- *A discussion of practical matters* that have to be resolved to apply this scheduler in real-life systems; and
- *A secure real-time resource access protocol* to share resources in an information-flow-secure manner.

1.3.2. A Sound Security Type System for Low-Level Operating-System Code

The second central contribution of this thesis is a control-flow-sensitive security type system for the low-level operating-system code of microkernel-based systems.

The principles for provable operating-system security go back to the mid 70th [FN79, BL73, FLR77]. Recent approaches to formally verify the absence of security policy violating information flows are typically instantiations of Rushby's non-interference framework [Rus92]. A proof in this framework involves proving two unwinding properties for all atomic transitions that a system can make.

However, although extensions of Rushby's framework have successfully been applied to an access-control mechanism [Rus92], to a multi-applicative smart card [SRS+02], to the Infineon SLE66 smart card processor [vOWL03], and to an abstract Haskell model of an L4 microkernel [LEA07, Les06], none of these approaches establish non-interference for a concrete implementation. As experienced by Kemmerer and McHugh [HKMY87], the lack of automation, the difficulty of identifying covert channels from failed proofs, and the complexity of the proofs themselves result in significant costs for verifying non-interference at the source-code level. The seL4 verification [KEH+09] has shown that confidentiality-preserving refinement proofs [HPS01], which connect properties of an abstract model to a concrete implementation, are principally feasible for modern high-performance microkernels[6]. However, the costs of such a proof are significant.

Security type systems are both easily automated and they avoid the costs of source-level non-interference proofs. However, to apply these analyses to low-level operating-system code, several challenges have to be mastered. As we shall see in greater detail in Section 4.2, such an analysis has to cope:

- with a combination of C++, C, and Assembler code;

[6] The refinement proof for seL4 is limited to Hoare properties [BT82, Jac89]. Non-interference is not preserved under these refinements.

- with interactions between the checked program, its clients, the kernel, and other servers;
- with peculiar execution environments and peculiar interactions with the underlying hardware;
- with code that exhibits non-deterministic behavior in these environments; and,
- with code for which the security policy is typically not completely known at the time of the analysis.

In part, these challenges have been addressed before [Sab01a, SM02, OCsC06, RS06]. However, as we shall see in greater detail in Section 4.3, contemporary language-based information-flow analyses for low-level operating-system code result in rather complex and unmaintainable security type systems when these type systems should check the microkernel or a multilevel server in its entirety. To avoid this complexity, the present work follows a different approach, which was originally suggested by Furuse et al. [FDKHN07].

The approach followed in this thesis is to first translate the C++ operating-system code into an intermediate programming language and to then check the translated program with a security type system for this intermediate language. Furuse et al. apply Gimple [SC08a], the intermediate language of the Gnu Compiler Collection. However, Gimple depends on a specific compiler and the translation from C++ to Gimple is not trivial. Hence, for an all-embracing compiler-independent soundness result, the translation from C++ to Gimple must be shown to preserve the semantics and information-flow properties of the checked C++ programs. To remain compiler independent and to avoid the costs of these refinement proofs, I introduce a new intermediate language called *Toy*.

Toy stays as close as possible to the C++ standard while providing the language constructs required to address the above challenges. For example, to formalize the non-deterministic evaluation order of C++ expressions [PC09, § 1.9 pt 13], *Toy* contains several non-deterministic composition statements to explicitly state the interleaving of C++ side effects and value computations.

After the to-be-checked C++ operating-system code is translated into a corresponding *Toy* program, this *Toy* program is complemented with subprograms, which describe the side effects that interactions with the underlying hardware ensue. Because both, the C++ operating-system code and these interleaved-executing side effects are formalized in *Toy*, a security type system for this intermediate language can check both for the absence of security policy violating information flows.

The detailed contributions that this thesis makes in the area of static information-flow analyses for low-level operating-system code are:

- *The non-deterministic intermediate programming language Toy*;
- *A control-flow-sensitive security type system for the deterministic part of Toy*;
- The notion of *learned secrets* to track the secrecy level of information that concurrently executing threads may learn from the checked program; and
- *A machine-checked soundness proof of the proposed security type system.*

Although, *Toy* is inherently non-deterministic, the proposed security type system focuses only on the deterministic core of *Toy*.

The reasons for that are twofold:

1. the standard typing rules for non-deterministic composition (see e.g., Sabelfeld [Sab01b, pg. 45]) apply, although, as we shall see in Section 4.7.1, at a loss of precision; and,

2. because *Toy* clearly separates input non-determinism from control-flow non-determinism, there is an alternative to applying the standard typing rules to all occurrences of the latter type of non-determinism: the security type system can check all possible ways in which the control-flow non-determinism in selected parts of the program can be resolved.

The key benefit of the latter approach is that the checked program p must not automatically be rejected as being potentially insecure if some resolutions of the non-determinism in p are potentially insecure. Often, a failure to check all these parts merely limits the safe compiler optimizations on p respectively the hardware platforms on which p can safely be run.

I have demonstrated the applicability of the security type system for *Toy* in three case studies:

- a memory access, which causes the hardware to walk through the page tables for the accessed virtual address;

- an L4-IPC send operation; and

- a buffer-cache server, which multiplexes the memory pools of its clients to cache recently-accessed file blocks.

A proof that Osvik's countermeasure protects against AES cache-side channel attacks complements these case studies. In this proof, I exploit an important property of the control-flow-sensitive security type system for *Toy*, which I shall introduce in Chapter 4: the ability to tolerate temporary breaches of confidentiality in lock-protected shared-memory regions. As long as all threads adhere to the locking discipline, lock-protected shared-memory regions may temporarily reveal information about confidential data (here the encryption key) as long as the checked program repairs this breach of confidentiality (Osvik's countermeasure) before it releases the protecting lock. In Osvik's countermeasure, the protecting lock is to disable processor interrupts until both the encryption round and its accompanying countermeasure completes. The disabling of interrupts automatically enforces adherence to the locking discipline because it prevents other threads from running on the same CPU and hence from deducing the AES key from cache conflict misses before Osvik's countermeasure has eliminated this possibility.

1.4. Scope and Limitations

Although the results of this thesis have a much broader scope, this thesis focuses on three main research areas:

1. perfect information-flow security,

2. L4-family microkernels, and

3. security type systems.

In the first part of this section, I discuss alternative directions and give reasons for my decision to focus on the above areas. In the second part, I discuss the assumptions on which my solutions are based and the limitations they have.

This thesis leaves the construction of an efficient type checking tool for low-level kernel code as future work. In principle, it is well known how sound security type systems translate into such tools [Mye99, Sim03], even for control-flow-sensitive analyses [FTA02, HL09].

1.4.1. Perfect Information-Flow Security

It is a common believe that realistic covert-channel-free systems cannot be built. Nevertheless, I strive in this thesis for a complete absence of illegal information flows over software-centric covert channels. The following five points motivate this decision.

1. Whether a system can tolerate low-bandwidth covert channels, and if so, how many bits per second are tolerable, depends on the type of confidential data a system has to protect. Unfortunately, size and value of confidential data are not always positively correlated [DS09].

   For example, in many systems, encryption keys are the most valuable data because they inherit their value from the confidential data they have encrypted. Yet, recommended key sizes for long-term (i.e., pre quantum computer) protection are only 128 bits for symmetric cryptography and 3248 bits for asymmetric cryptography [II09]. Given these numbers, I have to agree with J. Millen [Mil99] when he asks in his panel speech “20 years of covert channel analysis”: “how long is that [key] going to be kept secret even at one bit per second?”. Approximately two minutes for symmetric keys respectively little less than one hour for asymmetric keys are quite short periods, even if we neglect that knowledge of only a few key bits significantly improves attacks on the cipher (see e.g., Nohl [Noh08]).

2. The continuing increase of processor speed results in an increase of covert-channel bandwidths. As a result, capacity tradeoffs, which were justified for one processor generation, may no longer be justified on newer processor generations. Constant costs arise for re-evaluating channel bandwidths and for re-engineering those parts of the system where bandwidth-reduction schemes fail to sufficiently mitigate a covert channel.

   With the exception of the envisaged timing-leak transformations for low-level operating-system code, none of the solutions, which I shall propose in this thesis, depends on the speed of the underlying processor. For the timing-leak transformation, it suffices to update the safe worst-case execution-time estimates of secrecy-dependent operations. We shall return to this point in Section 2.4.6 on page 32.

3. Both, the machine-checked proof of the budget-enforcing fixed-priority scheduler and the machine-checked soundness result of the *Toy* security type system establish non-influence [Ohe04], an extension of Meseguer’s and Goguen’s non-interference property [GM82]. However, neither non-interference nor non-influence is prepared to tolerate the leakage of even as few information as a single bit. Hence, these properties hold only for perfectly secure systems.

   Approximate non-interference [DPHW02] and quantitative non-interference properties [RD82, CHM02, Low02, BP03] tolerate low-bandwidth information flows. However, they are much more difficult to establish.

4. Perfectly secure systems nicely combine with systems that tolerate low-bandwidth covert channels. That is, if our application scenario permits low-bandwidth covert channels, it can still be run on top of a perfectly-secure microkernel-based system. And,

5. Finally, it is an interesting research question to see how far one can get with perfect security in open microkernel-based systems. And perhaps, it is even possible to build covert-channel free systems [Mil99, PN92].

1.4.2. L4-Family Microkernels

Although the results of this thesis are also applicable to other microkernels and to other microkernel-based systems, I focus in this thesis on systems based on L4-family microkernels [Lie95, Hoh02, KV05, DdEE, WL10, Ste09a]. My choice for this particular kernel family is motivated as follows.

1. L4-family microkernels have demonstrated their ability to co-host potentially untrustworthy legacy operating systems and their applications next to security-sensitive and real-time-critical applications [HBB+98, HHF+05]. Hence, any modification must preserve both this co-hosting capability and the real-time capabilities of the microkernel;

2. L4-family microkernels implement one of the fastest possible IPC paths and they are highly optimized for performance. Kernel modifications must therefore preserve the performance of these kernels to the best degree possible; and,

3. Having contributed to the design and implementation of one of these kernels myself [KV05], I am familiar with the microkernels of the L4-family and with most of the peculiar programming patterns they contain.

I do not expect any difficulties in adapting the results of this thesis to other microkernels.

1.4.3. Security Type Systems and Static Analyses

Besides security type systems, there are several other static and dynamic approaches to language-based information-flow security (see e.g., [AB04, AB07, LUV05, Zan02, FM06, ML97]). The focus of this work is on static information-flow analysis and, more precisely, on security type systems for low-level operating-system code. In Section 2.4, I relate my approach to abstract-interpretation-based, data-flow-based, and logic-based information-flow analyses.

Dynamic approaches observe the information flows in a system [VEK+07, ZBWKM06, KYB+07] or in an application [ML97] and stop the system if information is about to be leaked. There are two reasons why dynamic information-flow security cannot be applied to all applications of open microkernel-based systems:

1. Without hardware support [TWM+09], the overhead of tracking secrecy levels of processed information can be significant. In particular, in the performance-critical IPC path such an overhead may be fatal. On the other hand, only a small fraction of the code of an open microkernel-based system needs to be constrained by such an analysis. Static analyses have no such overhead.

2. Real-time systems have to guarantee at admission time that real-time-critical applications will meet their deadlines. To give such a guarantee with a dynamic information-flow analysis, the potential occurrence of illegal information flows must be excluded at admission

time. However, because admission tests are typically run off-line, the analysis, which determines whether such an information flow occurs, must be static.

1.4.4. Assumptions

With the exception of cache side-channels in the proof of Osvik's countermeasure in Section 5.4, this thesis does not address hardware-centric covert channels. As we shall see in Section 2.1.3, hardware-based solutions are often more effective and more efficient in avoiding leakage over these channels. I will therefore assume that the envisaged open microkernel-based systems apply these hardware-based solutions.

In this thesis, I present various proofs that have been machine checked with the help of the interactive theorem prover PVS [ORS92]. The correctness of these proofs depends on the validity of the usual assumptions on the correctness of the underlying system. This includes the correctness of the theorem prover, of the operating system, of the programming environment, and of the underlying hardware platform. The PVS sources are published [Völ10].

In Section 3.3, I state further assumptions about the budget-enforcing fixed-priority scheduler. These assumptions are in part lifted in later sections in Chapter 3. Section 4.4 summarizes my assumptions about low-level operating-system code.

1.5. Synopsis

The remainder of this thesis is structured as follows: in the next chapter, I introduce the foundations of this work and relate my results to the works of others. Chapter 3 presents the budget-enforcing fixed-priority scheduler, the *ReThMo* task model, and the machine-checked non-interference proof for this scheduler. Chapter 4 introduces the *Toy* intermediate programming language, the security type system for low-level operating-system code in microkernel-based systems and its soundness proof. In Chapter 5, I apply the information-flow analysis of Chapter 4 in three case studies and in the correctness proof of Osvik's countermeasure against AES cache side-channel attacks. Chapter 6 concludes this thesis.

2. Foundations and Related Work

To my best knowledge, this is the first attempt towards a security-type-system-based information-flow analyses for the low-level operating-system code of microkernel-based systems. Still, there a large body of work that relates to the topic of this thesis.

This chapter presents a survey of related work and introduces the foundations on which this thesis is based. It is organized as follows: Section 2.1 gives a brief overview on covert channels following a classification by Sabelfeld and Myers [SS99, SM03]. Section 2.2 surveys security policies and the lattice-based notation of these policies. Section 2.3 introduces *non-influence* [Ohe04], the confidentiality property, which I shall use in Section 3.4, in the machine-checked soundness proof of the security type system for *Toy*, and in Section 4.7.3.3, in the machine-checked proof that the budget-enforcing fixed-priority scheduler protects against scheduling-related covert channels. In Section 2.4, I introduce security type systems and discuss related static analyses. In Section 2.5, I give a brief overview on L4-family microkernels and sketch how servers of L4-based systems look like. Section 2.6 discusses related non-interference-secure schedulers. Section 2.7 concludes this chapter with a brief introduction to the theorem prover PVS [ORS92], which I have used for the machine-checked proofs of this thesis.

2.1. Covert Channels

Lampson [Lam73] was first to identify covert channels as a security concern. Following the Trusted Computer Security Evaluation Criteria (TCSEC) [Gal93], I introduced in Section 1 covert channels as "communication channels that allow threads to transfer information in a manner that violates the system's security policy." In this section, I refine this definition for specific types of covert channels and give examples of source-level illegal information flows.

2.1.1. Storage and Timing Channels

Kemmerer [Kem83] classifies covert channels into *covert storage channels* and into *covert timing channels*. Storage channels are sender modifiable attributes of explicitly or implicitly shared resources (e.g., the free space on a shared disk). Receivers can directly or indirectly read the changed attribute (e.g., in the form of a 'disk full' error message). Timing channels reveal an attribute change or a resource usage indirectly through variations in the response times of receiver initiated operations.

Scheduling-related covert channels [SGLS77] are covert timing channels, where the sender varies its scheduling behavior with the intention that the scheduler reflects this variation in the points in time when it runs the receiver. They are also called *external timing channels* because the receiver is typically not cleared for sanitized sender outputs. Hence, it can observe only the externally visible runtime behavior of the sender.

In contrast to external timing channels, *internal timing channels* reveal the points in time when legitimately observable outputs of the sender occur. Provided a program has been successfully checked for the absence covert storage channels, a sound timing-leak transformation [Aga00a] eliminates both external and internal timing channels. The budget-enforcing fixed-priority scheduler, which I shall introduce in Chapter 3, prevents also unchecked and thus potentially malicious programs from leaking confidential information over external timing channels.

2.1.2. Noise

If only the leaking program can write to a covert channel, this channel is said to be *noiseless*. Otherwise, if other programs can also write to this channel, it is said to be *noisy*. Striving for perfect information-flow security, I have to regard also noisy channels as potentially harmful.

2.1.3. Hardware- and Software-Centric Covert Channels

In Section 1, I have distinguished *hardware-centric covert channels* from *software-centric covert channels*. Examples of hardware-centric covert channels include cache side channels (see e.g., [Ber04]), timing channels from disk-arm movement [KC91] and covert channels that signal information through the processor's power consumption [KJJ99], emitted radiation [Age72, LU02], or heat. These channels have in common that secrets are encoded in certain hardware attributes that are not necessarily visible at the architectural level. In contrast to hardware-centric channels, software-centric covert channels encode secrets in architecturally visible attributes.

Hardware-centric covert channels are, to a large degree, beyond the control of software-based solutions. Cache coloring [LHH97] forms an exception. However, even if software-based countermeasures mitigate these channels, hardware-based countermeasures [WL07, Age94, Gra93] are often more effective and more efficient.

With the exception of cache side channels in the proof of Osvik's countermeasure for AES [OST05] in Section 5.4, I do not address hardware-centric channels in the present work. I shall assume instead that hardware-based countermeasures are applied to mitigate the effects of these channels.

2.1.4. Illegal Information Flows in Source Code

All widely used methods for identifying covert channels in source code [Gal93] (security type systems included) are based on identifying illegal information flows. For that, the security policy of the system is broken down to program variables (e.g., based on the clearance of observers to which such a variable may eventually become visible or based on initially stored secrets). The secrecy levels of assigned-to variables are required to dominate the secrecy levels of all those variables that cause information to flow into such a variable.

Sabelfeld and Myers [SS99, SM03] give the following informal classification of source-level illegal information flows.

Explicit information flows arise when programs store and keep secrets in variables that eventually become visible to an observer that is not cleared for this information.

An example of an explicit flow is the assignment [1]:

```
l = h;
```

Implicit flows arise when programs leak secrets through their control flow.

An example of such a flow is

```
if (h % 2) { l = 1 } else { l = 0 }
```

It leaks the least-significant bit of **h** by assigning different values to **l**.

Internal timing leaks arise when programs encode secrets in the timing information of legitimately-observable events.

For example, the following program leaks the least-significant bit of **h** in the time when it sets **l** to one. The statement **idle(n)** stands for a no-op that lasts n μs.

```
l = 0;
if (h % 2) {
  idle (100);
}
l = 1;
```

External timing leaks arise when programs encode secrets in their execution and blocking behavior.

For example, the following program leaks the least-significant bit of **h** because it blocks if ***h % 2 == 1*** holds (**sleep**) and executes (**idle**) otherwise. In contrast to **idle**, **sleep** releases the CPU and permits the scheduler to run other threads in the meantime.

```
if (h % 2) {
  sleep(50);
} else {
  idle (50);
}
l = 1;
```

The time after which the above program sets **l** to one is $50\mu s$ regardless of the branch it takes. Hence, it contains an external timing leak but no internal timing leaks.

External timing leaks can be used to send confidential information over scheduler-related covert timing channels.

Termination leaks arise when programs encode secrets in the time when they terminate. Because non-termination and very long execution can typically not be distinguished by an external observer, termination leaks can be viewed as a form of external timing leaks.

An example of such a leak is

```
if (h % 2) {
  while(true) {}
}
```

[1] Like before, **h** is a variable that stores *high*-classified information. The *low*-classified observer of the variable **l** is not cleared for this information.

Probabilistic leaks arise when programs encode secrets in the probability distribution of observable outputs.

For example, the following program leaks the least-significant bit of **h** because the probability that **l == 1** is 100 % if **h == 1** and 50 % if **h == 0**. The expression **random(0...1)** returns each of the two values zero and one with the same likelihood.

```
if (h % 2) {
  l = 1;
} else {
  l = random(0...1)
}
```

Lowe [Low04] addresses a further class of information flows:

Refinement leaks arise when a concrete implementation of the checked program resolves the same non-deterministic choice in different ways. For example, the following program leaks the least-significant bit of **h** if a concrete implementation resolves the non-deterministic choice $1 \,[]\, 0$ in the if-branch in favor of 1 and in the else-branch in favor of 0.

```
if (h % 2) {
  l = 1 [] 0;
} else {
  l = 1 [] 0;
}
```

The timing- and termination-insensitive security type system for the deterministic core of *Toy*, which I shall introduce in Chapter 4, checks the low-level operating-system code of microkernel-based systems for the absence of explicit and implicit information flows. A subsequent timing-leak transformation [Aga00a] eliminates harmful internal timing leaks. External timing leaks are addressed with the help of the budget-enforcing fixed-priority scheduler, which I shall introduce in Chapter 3. Because I assume that the individual invocations of the microkernel and of the multilevel servers terminate, termination leaks are a non-issue. Next, I introduce information-flow policies, the security policies of interest for this thesis, and their lattice-based notation.

2.2. Information-Flow Policies

An end-to-end protection of confidential data must not only be concerned about the release of confidential information but also about its propagation. However, the primary concern of access-control policies is to prevent only the release of information to unauthorized subjects. Information-flow policies seek to control also where released information propagates. The security policies of interest for this thesis are therefore information-flow policies.

Since the pioneering works of Bell and La Padula [BL73] and of Denning [Den76], information-flow policies are typically characterized by the *lattice model*. In this model, information-flow policies are described by triples $(L, \leq, dom)$, which consist of a finite set of secrecy levels L, a dominates relation $\leq$ and a domain dom. Intuitively, if $l_s \leq l_r$ holds for two secrecy levels, a subject e_r that is cleared to l_r (i.e., $dom(e_r) = l_r$) can see more sensitive (i.e., higher classified) information than a subject e_s that is cleared to l_s. In particular, e_r may receive any information from e_s but not necessarily vice versa.

Subjects are typically users or, more precisely, the programs that execute on their behalf. Objects are typically files. However, it is also possible to consider more fine grained subjects and objects such as server threads or program variables. Objects and subjects are collectively called *entities*.

The function dom assigns a secrecy level to each entity. This secrecy level is usually called the *domain* of this entity. The domain of a subject is typically the least upper bound of the secrecy levels of information that this subject may know. It is called the *clearance* of this subject. For objects, the domain is typically the least upper bound of the secrecy levels of information the object may store. It is called the *classification* of the object.

The dominates relation $\leq$ relates secrecy levels to characterize between which entities information may flow. Information flows from e_s to e_r are authorized if and only if $dom(e_s) \leq dom(e_r)$. Information must not flow from e_s to e_r if $dom(e_s) \not\leq dom(e_r)$. Two secrecy levels are *incomparable* if neither $dom(e_s) \leq dom(e_r)$ nor $dom(e_r) \leq dom(e_s)$ holds. In this case, information flow in any direction between the respective entities is forbidden.

2.2.1. Lattice and Non-lattice Models

In many information-flow policies, the set L and the relation $\leq$ form a lattice. The tuple $(L, \leq)$ is a lattice if $\leq$ is a partial order (i.e., reflexive [2], transitive [3] and antisymmetric [4]) and if all non-empty finite subsets $S \subseteq L$ have a least upper bound $\sqcup S$ and a greatest lower bound $\sqcap S$.

However, in practice, these restrictions are often relaxed. For example, Almeida Matos et al. [MB05] assume $\leq$ to be a preorder (i.e., reflexive and transitive but not necessarily antisymmetric); in Section 3.4.5, I shall require $\leq$ to be reflexive and $(\leq, L)$ to be uniquely bounded from above $\top$ and from below $\bot$. That is, $\exists \top \in L.\ \forall l \in L.\ l \leq \top \ \vee\ l = \top$ and $\exists \bot \in L.\ \forall l \in L.\ \bot \leq l \ \vee\ l = \bot$. In particular, $\leq$ needs not to be transitive.

2.2.2. Intransitive Information-Flow Policies

If $\leq$ is not transitive, the information-flow policy is said to be *intransitive*. The intuition behind intransitive information-flow policies [Rus92] is to authorize information to flow from l_s-classified entities to l_r-classified entities only if this information passes an l_m-classified subject, that is, $l_s \not\leq l_r$ but $l_s \leq l_m$ and $l_m \leq l_r$. The role of the l_m-classified subject is to monitor and sanitize the information it forwards.

Let me introduce two further terms to reason about intransitive information-flow policies. I say the triple of secrecy levels $(l_s, l_m, l_r) \in L \times L \times L$ is an *intransitive pass* if it holds that $l_s \leq l_m \ \wedge\ l_m \leq l_r \ \wedge\ l_s \not\leq l_r$. I call the secrecy level l_m in the middle of an intransitive pass the *intransitive point* of this pass.

2.2.3. Downgrading and Dynamic Policies

In general, information-flow policies are not entirely static. That is, both the dominates relation $\leq$ and the domain dom may change [5].

[2] $\forall x.\ x \leq x$

[3] $\forall x, y, z.\ x \leq y \ \wedge\ y \leq z \ \Rightarrow\ x \leq z$

[4] $\forall x, y.\ x \leq y \ \wedge\ y \leq x \ \Rightarrow\ x = y$

[5] I assume L has been chosen sufficiently large to avoid later changes. Note, this does not prevent concrete systems from storing only the used subset of L.

In the context of open microkernel-based systems, we have to distinguish two forms of dynamic policies:

1. Those that change the accessibility of kernel or server objects; and
2. Those that declassify information that is derived from an accessible object [MSZ06, SS05].

Examples of the first class include the immediate revocation of access rights by the system administrator (see for example [Age99, Section 5.4.7 - Revocation of User Attributes (FMT_-REV.1)]) and the *power box* [Sti00]. The powerbox is a mechanism through which users can specify the authority a program should assume. In Section 2.5 of this introduction, we shall see that a reconfiguration of L4's access-control mechanism suffices to change the accessibility of kernel or server objects in L4-based systems.

Two examples of the second class of dynamic policies are the automatic disclosure of military documents after the passage of a certain amount of time and after these documents have been sanitized [otAH88], and a password checker. The latter validates a given password against a secret password file. To reject invalid logins, it has to reveal the boolean result to potentially unauthorized users that the password is not contained in the password file. The password file remains inaccessible to the requesting client.

Although sufficiently-strong encryption protects the confidentiality of encrypted data, the release of the ciphertext is in many aspects similar to the declassification of secret data. The ciphertext is derived from the secret key and from the secret plaintext. Once the encryption completes, it is safe to reveal the ciphertext (e.g., to lower classified network- or storage servers).

Almeida Matos and Boudol [MB05] propose an elegant way to describe when to declassify information and which part of the checked program is authorized to do so. If a part c of a program p requires a temporarily-relaxed information-flow policy $(L, \leq', dom)$ to release confidential information, Matos annotates this subprogram with the flow directive:

flow$(\leq')\{$c$\}$

A subsequent static analysis then checks this subprogram against this adjusted information-flow policy $(L, \leq', dom)$. Once c completes, the original information-flow policy $(L, \leq, dom)$ is restored. The remainder of p must therefore obey the more restrictive original policy. A program that is secure with regards to these changing policies is said to be *non-disclosure* secure.

The following pseudo code exemplifies the use of flow directives to authorize the release of the password check.

```
bool h;
bool l;

h = check_password_file(user, passwd);
flow(high ≤ low){ l = h; }
```

In this example, the boolean **h** stores the $high$-classified information whether the pair **user**, **passwd** is stored in the password file. It is assigned to the low-classified variable **l**. The flow directive authorizes the information flow from $high$ to low only for this assignment.

The security type system for *Toy* cannot directly be used to check operating-system code that declassifies confidential information. However, Matos and Boudol [MB05] have been able to show that non-disclosure generalizes non-interference (see below) for programs that contain no declassification. To check declassifying operating-system code, we can therefore use the security type system for *Toy* to establish non-interference of the individual parts for which the information-flow policy stays constant and Matos' analysis to check whether these parts combine to a non-disclosure-secure program.

2.3. Non-interference

Non-interference [GM82] is the prevailing formalization to assert the complete absence of security policy violating information flows in deterministic systems.

A deterministic system is non-interference secure with regard to an l-classified observer if this observer cannot distinguish any two runs of the system that start from observationally-indistinguishable initial states. Two states are *observationally indistinguishable* for an l-classified observer if all actions a, whose effect this observer may see (i.e., $dom(a) \leq l$), produce the same observable outputs. An action a can thereby be "inputs", "commands", or "instructions" to be performed by the observed system.

For non-deterministic settings there are two predominant ways to formalize non-interference: non-interference-properties based on state automata [Rus92, Ohe04], and non-interference-properties based on process algebras [Low04, FG01] (such as Hoare's communicating sequential processes (CSP) [Hoa78] and Millner's CCS [Mil89b]).

In this work, I shall use instantiations of *non-influence* [Ohe04], a state-automaton-based non-interference property by David von Oheimb.

The marker scheme [Low04] by Gawin Lowe and the corresponding typing rule for non-deterministic composition is one way to address the non-determinism in low-level operating-system code. Lowe's marker scheme ensures that the same non-deterministic choice is always resolved in the same way. We shall return to this point in Section 4.5.3.3. In the next section, I introduce non-influence in greater detail and illustrate its relation to non-interference.

2.3.1. Non-influence

Non-interference is concerned with the secrets that actions (e.g., client invocations) introduce in the system (e.g., a server) and that are possibly observed via outputs. However, there are also scenarios (most notably language-based information-flow security) where the system must not leak initially present secrets. Although these initially present secrets can be encoded as action sequences which place these secrets into the initial state, such a formalization is quite unnatural. I therefore follow von Oheimb [Ohe04] and distinguish the leakage of initially present secrets and the leakage of secrets introduced by later occurring actions. The respective properties, which assert the absence of leakage, are *non-leakage* and *non-interference*. *Non-influence* [Ohe04] asserts both the absence of leakage of initial secrets and the absence of leakage as a side effect of interactions (e.g., of clients with a checked operating-system server). Hence, non-influence is non-leakage plus non-interference. In the following, I formally define non-influence for state-transition systems.

2.3.1.1. State-Transition System

State-transition systems are a natural way to formalize state-oriented systems. A state-transition system $(S, A, \rightharpoondown)$ is defined by a set of states S, a set of actions A that the system should perform, and a possibly partial and non-functional transition relation $\rightharpoondown$. I write $s^i \overset{a}{\rightharpoondown} s^{i+1}$ ($s^i, s^{i+1} \in S, a \in A$) to denote that the atomic action a yields the result state s^{i+1} when executed

on the state s^i. The action a may not be enabled in the state s^i. In this case, s^i may not appear on the left-hand side of $\xrightarrow{a}$. Moreover, if the execution of a is non-deterministic, more than one result state may appear on the right-hand side of $s^i \xrightarrow{a}$, one result state for each way in which the non-determinism in a can be resolved.
The relation $\xrightarrow{a}$ executes a single (atomic) step of the system. System runs are described by lifting steps over action traces $\alpha \in A^*$. They are defined by straightforward primitive recursion:

Definition 1. System Run

Given an initial state $s^0 \in S$, a run of the state-transition system $(S, A, \rightarrow)$ that starts from this initial state s^0 is defined as

$$s^i \xrightarrow{\epsilon} s^i$$

for the empty trace ϵ, and as

$$s^i \xrightarrow{a \circ \alpha} s^j := \exists\, s^{i+1}.s^i \xrightarrow{a} s^{i+1} \;\wedge\; s^{i+1} \xrightarrow{\alpha} s^j$$

for the trace $a \circ \alpha$, which starts with the action a and which continues with the action trace α.

Two actions a and b are *atomic* [Lip75] [6] with regard to each other if whenever a parallel execution of these actions on a state s produces a result state t, this result state can also be produced by one of the sequential compositions of these actions. That is, $s \xrightarrow{a\|b} t \Rightarrow s \xrightarrow{a;b} t \vee s \xrightarrow{b;a} t$ must hold for all s and t, where $\|$ describes parallel execution and ; is the sequential composition operator.

2.3.1.2. Formalization of Non-influence

A state-transition system is non-influence secure with regard to an l-classified observer if the outputs of any two runs of the system, which start from l-similar initial states and which execute l-similar action traces, are observationally indistinguishable for this observer. Two initial states s^0 and t^0 are l-similar if they agree on the values of $\leq l$-classified variables. Two action traces are l-similar if they agree on their $\leq l$-classified actions and on the order of these actions. Intuitively, if the l-observable results are identical despite variations in the higher or incomparably classified actions and despite variations in the values of higher or incomparably classified variables, none of these results can depend on the secret of these actions or variables.

To formalize non-influence, we first have to formalize the parts of the actions and of the states that an l-classified observer may see. Let $sources(\alpha, l)$ be defined as follows.

Definition 2. Sources

Given an action trace α and an observer secrecy level l, $sources(\alpha, l)$ is recursively defined as

$$sources(\epsilon, l) := \{l\}$$

for the empty trace ϵ, and as

$$\begin{aligned} &sources(a \circ \alpha, l) := \\ &\qquad sources(\alpha, l) \,\cup\, \{w \mid \exists v.\ dom(a) = w \;\wedge\; w \leq v \;\wedge\; v \in sources(\alpha, l)\,\} \end{aligned}$$

for the trace $a \circ \alpha$, where $dom(a)$ is the classification of the action a.

[6] Lipton calls this property linearizable.

Intuitively, $sources(\alpha, l)$ collects all the secrecy levels of all those actions that are authorized to pass information directly to l-classified entities or indirectly via servers that are trusted to properly sanitize the passed information. In the trace α, the actions of sanitizing servers are represented as subsequences of α that follow a possibly leaking action a. For example, assume $dom(a) \leq dom(sanitize_a) \leq l$, $dom(a) \not\leq l$ for two actions a and $sanitize_a$, then $sources(a) = \{l\}$ because $sources(\epsilon) = \{l\}$ and because a must not directly send information to l-classified entities. However, because $dom(sanitize_a) \leq l$ holds, $sources(sanitize_a) = \{dom(sanitize_a), l\}$. Therefore, if a precedes the sanitizing action $sanitize_a$ in the action trace α, $sources(a \circ sanitize_a) = \{dom(a), dom(sanitize_a), l\}$ holds. Because a is sanitized in α, it may indirectly pass information to l-classified entities.

Based on the definition of sources, we can now define the subsequences of action traces that an l-classified observer can directly or indirectly (after they have been sanitized) see:

Definition 3. IPurge
Given an observer secrecy level l and an action trace, the subsequences of this action trace that l-classified observers may directly or indirectly see is recursively defined as

$$ipurge(\epsilon, l) := \epsilon$$

for the empty trace ϵ, and as

$$ipurge(a \circ \alpha, l) := \begin{cases} a \circ ipurge(\alpha, l) & \textbf{if } dom(a) \in sources(a \circ \alpha, l) \\ ipurge(\alpha, l) & \textbf{otherwise} \end{cases}$$

for the trace $a \circ \alpha$.

Intuitively, an action a is observable by an l-classified observer if either a is cleared to send to this observer (i.e., if $dom(a) \leq l$ holds) or if the information flows from a have been sanitized by subsequent actions in the trace α. This is the case if $dom(a) \in sources(\alpha, l)$ holds.

Definition 4. l-similar Traces
Two traces α and β are observationally indistinguishable for an l-classified observer, that is, they are l-similar if they agree on the subsequences that this observer may see:

$$ipurge(\alpha, l) = ipurge(\beta, l)$$

Two states s and t are l-similar if they agree on those parts that an l-classified observer may legitimately see. Let $output(l, s)$ extract the observations an l-classified observer can make on the state s. Let further $\overset{l}{\sim}$ be a relation over states — the *unwinding relation* — such that if $s \overset{l}{\sim} t$ holds for two states s and t, then $output(l, s) = output(l, t)$. We say:

Definition 5. l-similar States
Two states s and t are observationally indistinguishable for an l-classified observer, that is, they are l-similar if it holds that:

$$s \overset{l}{\sim} t$$

For the definition of non-influence, we have to lift the unwinding relation $\overset{l}{\sim}$ to sets of secrecy levels: $s \overset{L}{\approx} t := \forall l \in L.\ s \overset{l}{\sim} t$.

We can now define non-influence as the observational indistinguishability of two executions with l-similar action traces and l-similar initial states:

Definition 6. Non-influence

Given an information-flow policy $(L, \leq, dom)$, a state-transition system $(S, A, \rightharpoondown)$ is non-interference secure with regard to this policy and with regard to an l-classified observer if it holds that:

$$\begin{aligned}\forall \alpha, \beta \in A^*, s^0, s^i, t^0 \in S.\ ipurge(\alpha, l) = ipurge(\beta, l)\ \wedge\ s^0 \overset{sources(\alpha,l)}{\approx} t^0\ \wedge\ s^0 \overset{\alpha}{\rightharpoondown} s^i \\ \Rightarrow \quad \exists t^j \in S.\ t^0 \overset{\beta}{\rightharpoondown} t^j \wedge output(l, s^i) = output(l, t^j)\end{aligned} \tag{2.1}$$

A state transition system $(S, A, \rightharpoondown)$ is non-interference secure for all observers if Equation 2.1 holds for all $l \in L$.

Non-influence is timing insensitive because it contains no model of the timing of actions in α and β. However, it is termination sensitive because if the execution of α on s^0 terminates in s^i, this termination must be paralleled by the execution of β on t^0. In Chapter 4, I shall use a termination-insensitive security property. In this property, the existence of a terminating state t^j appears as a precondition.

As demonstrated by von Oheimb [Ohe04], non-influence combines two further properties: By removing the first precondition ($ipurge(\alpha, l) = ipurge(\beta, l)$) from Equation 2.1, we obtain a security property that is merely concerned about the leakage of secrets that are present in the initial states s^0 and t^0. Von Oheimb calls this property *non-leakage*. By removing the second precondition ($s^0 \overset{sources(\alpha,l)}{\approx} t^0$), we obtain *strong non-interference* as introduced by Mc Cullough [McC90] and Ryan [Rya90].

To remain consistent with other published works, I will not follow von Oheimb's terminology in this thesis. Instead, I show in Section 3.4.5 and in Section 4.7.3 how the proven security properties relate to non-influence.

2.3.2. Cryptography and Non-interference

Non-interference [GM82] and likewise non-influence [Ohe04] assume adversaries with unlimited computing power. Hence, they cannot tolerate the release of encrypted secrets [7] as in:

```
l = encrypt(h, k);
```

Clearly, in order to decrypt **l**, an encryption of two different values in **h** must result in two different ciphertexts. The program is not non-interference secure because variations of values of the $high$ variable **h** cause variations of the low variable **l**. However, unless adversaries break the encryption, the confidentiality of the secret in **h** is protected. Given a sufficiently strong encryption algorithm, breaking the cipher is computationally hard.

A common approach to tolerate encryption in secure programs is to relax non-interference by limiting the computational power of observers [DPHW02, RD82, Low02, CHM02, BP03, AHS08]. For instance, Askarov et al. [AHS08] argues for a possibilistic computational non-interference property (CNI). According to CNI, a program is non-interference secure if the set of possible l-observable outputs remains the same despite variations of high inputs. Applied

[7] Intransitive non-interference and non-influence can only be used to enforce that secrets pass an encryption unit before they are released.

to cryptography this means that identity of ciphertexts is relaxed in favor of a low-equivalence relation between ciphertexts, which is required to fulfil the following two properties:

1. Any ciphertext produced by each plaintext-key pair must have a low-equivalent ciphertext for any other choice of plaintext and key, and

2. For any two plaintext-key pairs there exists ciphertexts that are not low equivalent.

The first property ensures the safe use of encryption, the second prevents occlusion. *Occlusion* is the problem of hiding information flows towards the analysis by treating all encrypted values as low equivalent. The following example demonstrates this point.

```
l1 = encrypt(h, k);
if (h % 2) {
  l2 = encrypt(h, k);
} else {
  l2 = l1;
}
```

If all encrypted values would be considered low equivalent, an information-flow analysis cannot detect the leakage of the least significant bit of **h**. In the above example, this bit is leaked in the equality / inequality of **l1** and **l2**. If the least significant bit of **h** is set, **l1** and **l2** compare unequal because typically encryption algorithms add a random value to the encryption to protect against chosen plaintext attacks.

In this work, encryption will play a less central role because the microkernel and many multi-level servers do not rely on encryption to protect secret information. For those that do, I envisage to handle encryption in a similar way as Matos et al. [MB05] handles declassification. For example, as in

```
tmp = encrypt(h, k);
flow_CNI{l = tmp;}
```

to emphasize that only for the assignment of the encryption result in **tmp** to **l**, a relaxed non-interference property (e.g., CNI) should hold. Other low output variables have to fulfil a stronger non-interference property. An elaborative discussion of this point is out of the scope of this thesis.

2.3.3. Unwinding

Contemporary approaches to prove the absence of security policy violating information flows in operating-system kernels typically instantiate non-interference frameworks. In these frameworks, non-interference follows from a proof that two unwinding properties hold for all atomic steps of the kernel model. For non-influence, these unwinding properties are:

Definition 7. (uniform) Step Consistency
Given an information-flow policy $(L, \leq, dom)$, an atomic step $a \in A$ of the state transition system $(S, A, \rightharpoonup)$ is uniform step consistent for this policy if it holds that:

$$\begin{aligned} &\forall U \subseteq L, s^i, s^{i+1}, t^j \in S. \\ &\quad \exists u \in U.\ dom(a) \leq l \ \wedge\ s^i \xrightarrow{a} s^{i+1} \ \wedge\ s^i \overset{U \cup dom(a)}{\approx} t^j \ \Rightarrow \\ &\quad\quad \exists\, t^{j+1} \in S.\ t^j \xrightarrow{a} t^{j+1} \ \wedge\ s^{i+1} \overset{U}{\approx} t^{j+1} \end{aligned}$$

Definition 8. (uniform) Local Respect

An atomic step $a \in A$ of the state transition system $(S, A, \rightharpoondown)$ locally respects the unwinding relation for the information-flow policy $(L, \leq, dom)$ if it holds that:

$$\begin{aligned}
&\forall U \subseteq L, s^i, s^{i+1}, t^j \in S. \\
&\quad (\forall u \in U.\ dom(a) \not\leq u) \ \wedge\ U \neq \emptyset \ \wedge\ s^i \overset{U}{\approx} t^i \ \Rightarrow \\
&\qquad (s^i \overset{a}{\rightharpoondown} s^{i+1} \ \Rightarrow\ s^{i+1} \overset{U}{\approx} t^j) \ \wedge\ (\exists t^{j+1}.\ t^j \overset{a}{\rightharpoondown} t^{j+1} \ \Rightarrow\ s^i \overset{U}{\approx} t^{j+1})
\end{aligned}$$

Step consistency says that if the atomic step a is directly or indirectly (through sanitizing servers) visible to an l-classified observer, then executing a on l-similar states ($s^i \overset{L \cup dom(a)}{\approx} t^j$) produces states that are l-similar as well ($s^{i+1} \overset{L}{\approx} t^{j+1}$). The first clause of the *goodness* property (Definition 32 on page 164), which I use in the soundness proof of the security type system for *Toy*, is similar to step consistency.

Local respect says that if a is not directly or indirectly visible to an l-classified observer, then the execution of a must not have any side effects that this observer can detect. That is, the result of executing a on s^i is l-similar to t and vice versa the result of executing a on t^j is l-similar to s. The second clause of *goodness* (Definition 33) is similar to local respect.

In this thesis, I make no use of the two unwinding properties in Definition 7 and in Definition 8. Although I could have applied these properties for the non-interference proof of the budget-enforcing fixed-priority scheduler, I found the relation **same_high_state** (see Section 3.4.6 on page 95) to be a more intuitive invariant result for fixed-priority schedulers. Equivalence of l-observable outputs follows directly from this relation.

In Section 1.3.2, we have already seen four examples where unwinding properties have been used to prove non-interference of abstract models of an access-control mechanism [Rus92], of a multiapplicative smart card [SRS+02], of a smart-card processor [vOWL03], and of the IPC path of an L4-family microkernel [LEA07]. However, because these properties have to be established for all atomic steps, unwinding-based source-level verifications come at significant costs [HKMY87]. Security type systems and related static analyses avoid these costs. For example, the soundness proof of the security type system for *Toy* (in Section 4.7.3 on page 163) automatically establishes non-interference for all successfully checked *Toy* programs.

2.4. Security Type Systems and Related Static Information-Flow Analyses

Contemporary source-level security type systems typically check high-level languages such as Java [ML98, Str03], Caml [PS03] or Haskell [LZ06]. OS developers, however, require memory-management and data-structure controls that these languages do not provide [Sha06]. As a result, most operating systems are still written in a combination of C++, C and Assembler.

In this thesis, I focus on the low-level language features of C and C++, that is, on the representation of data types in memory and on memory accesses. Kernel programmers typically use high-level language features such as classes, inheritance, exceptions, and dynamic casts only very reluctantly because some of these features have considerable overheads and because others require non-trivial run-time support. The fundamentals for checking information flows that arise from these high-level features are well known.

$$
\begin{array}{lcc}
\text{[E1-2]} & M \vdash e : \mathit{high} & \dfrac{v \in \mathit{Vars}(e) \Rightarrow M(v) = \mathit{low}}{M \vdash e : \mathit{low}} \\[2ex]
\text{[C1-2]} & \dfrac{l_{ip} \leq M(v) \quad M \vdash e : M(v)}{[l_{ip}], M \vdash v := e} & \dfrac{[l_{ip}], M \vdash c1 \quad [l_{ip}], M \vdash c2}{[l_{ip}], M \vdash c1; c2} \\[2ex]
\text{[C3]} & \multicolumn{2}{c}{\dfrac{M \vdash e : l_{ip} \quad [l_{ip}], M \vdash c1 \quad [l_{ip}], M \vdash c2}{[l_{ip}], M \vdash \textbf{if } e \textbf{ then } c1 \textbf{ else } c2}} \\[2ex]
\text{[C4, S]} & \dfrac{M \vdash e : l_{ip} \quad [l_{ip}], M \vdash c}{[l_{ip}], M \vdash \textbf{while } e \textbf{ do } c} & \dfrac{[\mathit{high}], M \vdash c}{[\mathit{low}], M \vdash c}
\end{array}
$$

Figure 2.1.: Control-flow insensitive security type system for a simple imperative language.

An alternative to source-level analyses are assembly-level security type systems [SA98, AR05, ABR04]. Because they check the compiled binary respectively the bytecode for harmful information flows, they check also security-policy violations introduced by the compiler. However, unless all optimizations are disabled, a presumably non-interference-secure program does not necessarily result in a non-interference-secure binary [ABR04].

To remain independent from a specific compiler and, to a certain degree, also from a specific hardware architecture, I focus on source-level information-flow analyses. However, because checks are for non-deterministic *Toy* programs that origin from a translation of C++ programs, the analysis will check also those compiler optimizations that are described by the compiler-resolved non-determinism in these *Toy* programs. That is, the produced binary is non-interference secure if the compiler-resolved non-determinism in the successfully checked *Toy* programs can be resolved in such a way that both programs exhibit the same behavior.

2.4.1. Control-Flow-Insensitive Security Type Systems

The simplicity of the check and hence the performance of the analysis is the reason why most of today's security type systems are control-flow insensitive. A control-flow-insensitive type system seeks to infer the type of a program from the types of its subprograms. The type of a program is a security level, which summarizes the effects its statements and expressions have on the system state.

Figure 2.1 presents the typing rules of a control-flow-insensitive security type system for a simple imperative programming language and for the two-level lattice with $\mathit{low} \leq \mathit{high}$ and $\mathit{high} \not\leq \mathit{low}$. The type system is similar to the one Volpano and Smith present in [SV98].

A typing judgement has the form $[l_{ip}], M \vdash p$. It reads: the program p is typed in the typing environment M and in the context secrecy level l_{ip}. Soundness of this type system asserts non-interference for all typeable programs.

The typing environment M maps each variable v of p to the secrecy level of the information that is stored in v respectively to the clearance of observers of v. If the secrecy level of the context of p is l_{ip}, p cannot be typed if it writes any secret information to variables that are lower classified than l_{ip} (Rule C1). In other words, side effects of p are limited to higher or equally classified variables v (i.e., $l_{ip} \leq M(v)$). The Rules C3 and C4 check for implicit information flows by requiring $M \vdash e : l_{ip}$ for the condition e of the if-statement and of the while-statement. That is, in order to apply these rules, the context secrecy level l_{ip} must first be

$$[\text{E}]\quad \frac{l_{res} = \bigsqcup_{v_i} M(v_i) \quad v_i \in Vars(e)}{M \vdash e : l_{res}}$$

$$[\text{C1}]\quad \frac{M \vdash e : l_{res}}{[l_{ip}] \vdash M\ \{v := e\}\ M[v \mapsto l_{res} \sqcup l_{ip}]}$$

$$[\text{C2}]\quad \frac{[l_{ip}] \vdash M\ \{c_1\}\ M'' \quad [l_{ip}] \vdash M''\ \{c_2\}\ M'}{[l_{ip}] \vdash M\ \{c_1; c_2\}\ M'}$$

$$[\text{C3}]\quad \frac{M \vdash e : l_{ip} \quad [l_{ip}] \vdash M\ \{c_i\}\ M'_i \quad i \in \{1,2\} \quad M' = M'_1 \sqcup M'_2}{l_{ip} \vdash M\ \{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ M'}$$

$$[\text{C4}]\quad \frac{M \vdash e : l_{ip} \quad l_{ip} \vdash M\ \{c\}\ M}{l_{ip} \vdash M\ \{\textbf{while}\ e\ \textbf{do}\ c\}\ M}$$

$$[\text{S}]\quad \frac{l_1 \vdash M_1\ \{c\}\ M'_1}{l_2 \vdash M_2\ \{c\}\ M'_2} \qquad l_2 \leq l_1,\ M_2 \leq M_1,\ M'_1 \leq M'_2$$

$$[\text{C4'}]\quad \frac{M'_i \vdash e : t^i \quad [l_{ip} \sqcup t^i] \vdash M'_i\ \{c\}\ M''_i \quad 0 \leq i \leq n}{[l_{ip}] \vdash M\ \{\textbf{while}\ e\ \textbf{do}\ c\}\ M'_n} \qquad \begin{array}{c} M'_0 = M,\ M'_{i+1} = M''_i \sqcup M, \\ M'_{n+1} = M'_n \end{array}$$

Figure 2.2.: Flow sensitive security type system.

raised to the secrecy level of this expression. The subsumption rule (Rule S) fulfils this task.

The preconditions in the antecedent (above the line) of the typing rules of security type systems are typically limited to typing judgements for subexpressions or substatements and subtyping judgements [8] such as $l_{ip} \leq M(v)$. Constraint-based type systems [PS03] and type systems with existential types [MP85] allow also variables, constraints, and existential quantifiers as preconditions. However, as long as the security type system is control-flow insensitive, it cannot tolerate temporary breaches of confidentiality.

2.4.2. Control-Flow-Sensitive Security Type Systems

Figure 2.2 shows a control-flow sensitive security type system for the same imperative programming language. The typing rules origin from Hunt et al. [HS06].

Typing judgements of control-flow-sensitive security type systems have the form: $[l_{ip}] \vdash M \{ p \} M'$. In addition to the context secrecy level l_{ip}, the typing judgements take two typing environments M and M'. The typing environment M denotes the variable-to-secrecy-level mapping before p executes. That is, it holds the secrecy levels of information that is initially stored in the variables that p access. M' denotes these secrecy levels after p terminates. Hence, $[l_{ip}] \vdash M \{ p \} M'$ describes how p evolves the secrecy levels in M when executed in an l_{ip}-classified context.

In the typing judgements of control-flow-sensitive type systems, both M and M' appear on the right-hand side of $\vdash$. This is to reflect their changing when the typing rules decompose

[8] Note Rule E2 abbreviates a set of typing rules for the subexpressions from which e is composed. Hence, the precondition $v \in Vars(e) \Rightarrow M(v) = low$ translates into a subtyping judgement $M(v) \leq low$ for $M \vdash read(v) : low$ and into typing judgements for the subexpressions of e. An example of the latter is the rule: $\frac{M \vdash e_1 : low \quad M \vdash e_2 : low}{M \vdash e_1 + e_2 : low}$

p into its substatements and subexpressions. For example, if p is sequentially composed of c_1 and c_2, Rule C2 requires c_1 and c_2 to be types as $[l_{ip}] \vdash M \{ c_1 \} M''$ respectively as $[l_{ip}] \vdash M'' \{ c_2 \} M'$ where M'' denotes the variable-to-secrecy-level mapping after c_1 terminates and before c_2 starts. In contrast to control-flow-insensitive security type systems, the typing results of a previous occurrence of these substatements cannot be reused unless the typing environments have been identical. Therefore, because recurring substatements and subexpressions have to be reevaluated more often, the costs of flow-sensitive analyses are typically much higher [FTA02].

Initially, M maps each variable to an upper bound of the secrecy levels of the information that this variable holds in the initial states. M' initially denotes the observer clearances of the variables of p. In the course of typing p, the typing rule for the assignments in p (Rule C1) and the subsumption rule (Rule S) change these environments. If $v := e$ is an assignment that occurs in p, Rule C1 sets the secrecy level $M'(v)$ to the least upper bound of l_{res} and l_{ip}. Thereby, l_{res} is the secrecy level of the expression result e and l_{ip} is the secrecy level of the context in which the assignment appears. If this least upper bound is not dominated by the clearance of v, a control-flow-insensitive security type system would immediately reject the program p containing this assignment. A control-flow-sensitive type system can however tolerate this imminent breach of confidentiality as long as the actual breach is repaired before p terminates respectively before v becomes visible to lower or incomparably classified observers. In the above type system, the latter is assumed to occur only after p terminates. In the security type system for *Toy*, I shall lift this restriction.

Like before, the result of the conditions of if- and while-statements must be typed at l_{ip} in order to apply the typing rules for if (Rule C3) and for while (Rule C4). The subsumption rule allows to adjust both of the typing environments and the context secrecy level. More precisely, the secrecy levels in M and l_{ip} may increase but not decrease to assume higher classified information in the stored variables respectively a higher classified context. The secrecy levels in M' may only decrease to assume lower classified observers. A program that is secure in the presence of a lower classified observer remains secure if only higher classified observers can see the respective variables.

Rule C4' is an alternative typing rule for **while** [HS06]. It evaluates e and c until a *fixed point* $M'_{n+1} = M'_n$ is reached. Such a fixed point exists because the abstract-interpretation part of the typing rules is monotone (see [HS06, Theorem 4.1]). The abstract-interpretation part of the typing rules denotes how M' evolves from l_{ip} and M (see below).

2.4.3. Related Information-Flow Analyses

Although they origin from different theoretical backgrounds, control-flow-sensitive type systems [HS06], abstract-interpretation-based information-flow analyses [JPW05, Zan02] and Amtoft's and Banerjee's Hoare-like logic-based approach [AB04] show many similarities.

For example, the Rules E, C1, C2, C3 and C4' of the security type system in Figure 2.2 can also be found in abstract-interpretation-based information-flow analyses [JPW05, Zan02]. However, their interpretation is slightly different. Abstract interpretation (AI) symbolically executes a program on an abstract state [Cou96]. In the case of information-flow analyses, this state is the typing environment M enriched with the secrecy level of the "instruction-pointer", that is the context secrecy level l_{ip} and enriched with the secrecy level of expression results l_{res}.

The state (M, l_{ip}, l_{res}) is abstract because it keeps only the secrecy levels but not the concrete values of variables. The kept secrecy levels are upper bounds because the abstract-interpretation rules cannot detect whether a concrete expression cancels information in a variable. For example, **a = l + h – h;** clearly assigns only the low-classified information in **l** to **a**. However, because Rule E abstracts from the concrete values and from the concrete arithmetic operations in e, $M(a)$ is set to $M(a) = M(l) \sqcup M(h) = high$.

To reduce the complexity of the analysis, abstract-interpretation rules (and the Rules C3 and C4' of the above type system) typically recombine the results of alternative execution paths at so called *join points*. A join point exists after the branches of if-statements and after each iteration of while-statements. At the join point of an if-statement, control-flow-sensitive type systems and AI-based analyses combine the abstract states of the two branches M_1 and M_2 into the result state M' by taking the point-wise least upper bounds of their secrecy levels (i.e., $M' = M_1 \sqcup_{ptw} M_2$, where $M_1 \sqcup_{ptw} M_2 := \lambda v.M_1(v) \sqcup M_2(v)$ [9]).

The fundamental difference between control-flow-sensitive analyses and abstract-interpretation-based analyses is the absence of a subsumption rule in the latter. Hence, abstract-interpretation rules alone cannot check whether programs contain security-policy-violating information flows. Instead, abstract interpretation produces an abstract result state, whose secrecy levels must be checked against the observer clearances. Warnier [JPW05] calls this check $decreasing(M, M')$. It is defined as $\forall a.M(a) \leq M'(a)$.

The above type system by Hunt and Sands instantiated with the universal lattice $(\wp(Var), \subseteq)$ is equivalent to Amtoft and Banerjee's independence analysis [AB04]. This leads to information-flow analyses of information-flow policies that are not completely known at the time of the analysis.

2.4.4. A-Priori Unknown Information-Flow Policies

To check programs for security-policy-violating information flows, security type systems and related analyses typically require precise a-priori knowledge of the information-flow policy. However, because security policies are in general dynamic and because the microkernel and its multi-level servers can be reused in a variety of different systems, information-flow policies are to a large degree unknown at the time of the analysis. Consequently, it is not always possible to decide immediately whether information flows are harmful or benign. Still, it is interesting to identify all information flows and to record them for a later check once the precise information-flow policy is known.

In [AB04], Amtoft and Banerjee describe an information-flow analysis, which is based on a Hoare-like logic. In this analysis, non-interference is described through independence assertions of variables: $x \sharp y$. These assertions are the negation of Cohen's notion of dependency [Coh78]. A variable x is independent of y (written $x \sharp y$) if any two runs of the checked program, which agree in their initial states on the values of all variables except y, produce result states that agree at least on the value of x.

Given an information-flow policy $(L, \leq, dom)$ and an observer secrecy level l, a program with a set of independence assertions I is non-interference secure if for all ($high$) input variables x with $dom(x) \not\leq l$ and for all (low) output variables y with $dom(y) \leq l$ it holds that $x \sharp y \in I$.

[9]Where it is clear from the context, I shall write $M_1 \sqcup M_2$ instead of $M_1 \sqcup_{ptw} M_2$

Hunt and Sands [HS06] construct a control-flow-sensitive security type system based on Amtoft's and Banerjee's analysis. Independence assertions are thereby replaced by a universal lattice. This lattice consists of all subsets of variable identifiers $\wp(Var)$ and the partial order $\subseteq$. For example, the universal lattice for a two variable program is $(\{\{\}, \{l\}, \{h\}, \{l, h\}\}, \subseteq)$. To check data confidentiality, once the information-flow policy is known, the variable identifiers in the universal lattice are instantiated with the concrete secrecy levels of input variables.

For example, Hunt's control-flow-sensitive information-flow analysis of **l := l + h** returns $M'(l) = \{l, h\}$. A later instantiation with $dom(l) = low$ and $dom(h) = high$ gives $M'(l) = low \sqcup high = high$ if we assume the two level lattice with $high \not\leq low$, which reveals the leakage.

Laud et al. [LUV05] further substantiates the similarities between independence analyses and security type systems, which are based on universal lattices. He shows that certain type systems for sequential programs are equivalent to data-flow analyses. Laud instantiates two of these analyses to check information-flow security.

In this thesis, I extend the abstract-interpretation-based approach by Warnier et al. [JPW05] with a notion of shared memory and locks. However, I will formalize this approach as a control-flow-sensitive security type system.

To check low-level operating-system code whose information-flow policy is not entirely known at the time of the analysis, I follow Hunt and Sands' universal-lattice based approach. However, because interactions with operating-system code are not limited to the program start and termination, I have to extend this lattice to reflect when a shared-memory variable is read.

2.4.5. Operating-System Functionality

Security type systems typically abstract from the target underlying operating system. Sabelfeld [Sab01a], Mantel et al. [SM02], O Neill et al. [OCsC06], and Russo et al. [RS06] are exceptions to this rule. These works consider semaphores, blocking inter-process communication, communication channels and an interface to signal the underlying scheduler when it is safe to run low-classified threads.

However, as we shall see in greater detail in Section 4.3, the principle approaches of these works will not scale to the size and complexity of a microkernel or of a multi-level server. To prove data confidentiality of programs that invoke a certain operating-system mechanism, these works construct formal models of the respective OS functionality and specific typing rules to check the involved information flows. Finally, they prove the typing rules sound against the respective formal model of the checked OS functionality.

On the basis of a size-aligned virtual-memory read operation, we shall see that the typing rules of such a security type system tend to become rather complex and unmanageable. For this reason, I follow Furuse et al. [FDKHN07] and construct the non-deterministic intermediate programming language *Toy*. In *Toy*, interactions with the operating system and interactions with the underlying hardware appear as subprograms, which execute in an interleaved fashion with the translated C++ operating-system code. Both are subjected to the same information-flow analysis: the security type system for *Toy*.

2.4.6. Timing-Leak Transformations

Timing- and termination-insensitive security type systems, such as the two in Figure 2.1 and in Figure 2.2, risk overlooking internal and external timing leaks and leakages that encode information in the termination of the checked program. Nevertheless, the security type system for *Toy* is timing and termination insensitive.

Unexpectedly long lasting system calls or server invocations are typically quickly detected (though not as easily fixed). I will therefore not address termination leaks. Instead, I shall assume that the microkernel completes system calls in a bounded amount of time and that multi-level servers respond in a similar way to client requests.

To address internal timing leaks, I assume that the successfully checked operating-system code is subjected to a suitable timing-leak transformation. The budget-enforcing fixed-priority scheduler addresses external timing leaks.

Timing-leak transformations [Aga00b] are program transformations that produce timing-sensitive non-interference-secure programs from timing-insensitive non-interference-secure programs. To do so, they replace statements and expressions with secrecy-dependent timing behavior with semantically-equivalent statements and expressions that have no such secrecy-dependent timing behavior. Several such transformations have been proposed:

- *Cross copying* [Aga00a] copies the statements of both branches of an if-statement with secret conditional into the respective other branch. To preserve the semantics of the original code, it replaces assignments with equally-long lasting skip-statements.

- *Transactional branching* [BRW06] transforms the branches of if-statements with secret conditionals into transactions. The transformed program then executes the transactions of both branches. However, to preserve the semantics of the original code, only the transaction of the taken branch is committed.

- *Unification* [KM07] seeks to optimize the performance of *cross copying* by removing unnecessary `skip` statements. For that, unification identifies low-observable events in both branches and seeks to align identical events to occur in the same order and at the same point in time relative to the beginning of the branch.

In [BRW06], Warnier sketches a further, completely different approach, which is based on Engblom's worst-case execution-time (WCET) analysis [EES$^+$03]. Given a safe upper bound on the latest possible time when a low-observable event may occur, the transformation inserts a busy-waiting loop that defers this event to its safe upper bound.

With Engblom's method, timing-leak transformation is essentially reduced to a worst-case execution-time problem. The tighter the estimated worst-case bounds, the better the performance of the transformed program. Even unsafe bounds can be used if a concrete application scenario tolerates low-bandwidth covert channels. Moreover, besides having to access the system clock, the inserted code (though not the WCET bounds) is architecture and compiler independent. As a result, when binaries are shipped rather than source code, only the data section, which contains these WCET bounds, has to be patched to adjust the transformation to a new platform.

In a sense, the countermeasure to eliminate scheduling-related timing channels due to non-preemptive execution (see Section 3.3.5 on page 69) is such a timing-leak transformation.

2.4.7. Points-To Analysis

Static analyses for C and C++ programs immediately benefit from the results of two further types of static analyses: *points-to analysis* and *loop-bound analyses*.

Given a program p, a points-to analysis seeks to statically derive as precisely as possibly the addresses to which pointer variables may refer in a certain program state s of p. Examples of points-to analyses are [Wu, Ryd03, HL09, SWM00, WL02].

In this thesis, I will not integrate a specific points-to analysis into the security type system for *Toy*. Instead, I will assume that any correct points-to analysis is used to produce the pointer information the security type system requires. Let $pta(p)$ be a points-to analysis for p, which returns for each state s of p and for each pointer in p the set S of possible addresses to which this pointer may point to. The points-to analysis $pta(p)$ is correct if the returned sets S contain at least the actual address to which the respective pointer refers. S may however also contain other addresses if the precise pointer destination cannot be determined. There are two fundamental ways to react to imprecise points-to information:

1. In the analysis, we may pick one address at a time and check the remainder of the program under the assumption that this picked address is the actual address; or

2. If S contains more than one address, we can apply weak updates, as explained below, on all addresses in S.

Tlili and Debbabi [TD08] follow the first approach in their memory-safety analysis for C programs. In this check, they verify for the checked C program the absence of null-pointer dereferences, or accesses to deallocated objects, and the absence of reads to uninitialized objects.

To not risk overlooking information flows that involve reading or writing the actual pointer destination, we have to investigate all possible pointer targets. Hence, if we would follow the first approach in all situations, the analysis performance would deteriorate significantly. An efficient type checking tool must therefore select carefully when it follows the more precise first approach and where it reverts to weak updates.

Weak update [GMF79] is a safe approximation of writing through pointers whose destinations are not precisely known. The update is called weak because it would be unsafe to replace the information that a potentially targeted variable stores. In a control-flow-sensitive security type system, a weak update of a variable v in the set of pointer destinations S would therefore not only consider the secrecy level of the assigned expression but also the secrecy level of the information v holds before the assignment is evaluated. Hence, $M'(v) := l_{res} \sqcup l_{ip} \sqcup M(v)$.

The counterpart for a weak update is a *strong update*. Strong updates replace the previously-stored information completely. Therefore, they can only be applied to pointers whose destination is known precisely, that is, if the set S contains precisely one element. Such a set with precisely one element is called a *singleton set*.

Reads through pointers with imprecise pointer destinations work by returning the least upper bound of the secrecy levels of all possible destinations. That is, $l_{res} := \bigsqcup_{v \in S} M(v)$.

Static points-to analyses typically work with abstract addresses such as full-scope field or variable identifiers. However, because the virtual-to-physical address translation may ensue security policy violating information flows, these analyses are not immediately applicable to low-level operating-system code. A points-to analysis, which is applicable for an information-flow analysis of low-level OS code, must therefore return virtual addresses in the set of potential pointer destinations S. Wilson et al. [WL95] describe such an analysis for C / C++ pointer programs.

It reaps benefit of additional link-time information to deduce the potential virtual addresses of the referred objects.

2.4.8. Loop-Bound Analysis

A second type of static analyses, from which advanced static analyses for C / C++ programs benefit immediately, are loop-bound analyses (see e.g., [MAWF98, dMBCS08]). A correct loop bound analysis returns for each loop of the checked program p an upper bound on the number of iterations after which the loop is guaranteed to terminate.

Although the fixed-point iteration in Rule C4' of the control-flow-sensitive security type system in Figure 2.2 does not depend on such a bound to produce a safe approximation of the involved information flows, knowledge of such a bound can significantly improve the precision of the analysis. For example, the following **for**-loop sums up the first 5 elements of the smart array **a**.

```
smart_array<int> a[20];
int sum = 0;

a[6] = h;

for (int i = 0; i < 5; i++) {
  sum += a[i];
}

l = sum;
```

However, if we do not consider this bound, the analysis must pessimistically assume that all fields of the array are read. Hence, **sum** becomes $high$ because we cannot exclude an access to **a[6]**. For an ordinary array, which, unlike the smart array, does not limit accesses to the cells of the array, the results of a loop-bound analysis becomes even more important. This is because many C / C++ implementations translate out-of-bounds accesses into valid memory references to addresses beyond the array.

In this work, I shall assume that all system calls and server invocations terminate. Hence, there cannot be non-terminating loops in the checked pieces of operating-system code.

2.5. L4-Family Microkernels

Originating from Jochen Liedtke's initial design [Lie95], L4 has evolved into a family of microkernels [Lie96, Lie99, DLSU04, EHL98, Sch96, Hoh02, KV05, DdEE, Ste09a, WL10]. As second-generation microkernels, these kernels implement only two mechanism: IPC, and, in recent versions [KV05, WL10, DdEE], capabilities as the sole access-control mechanism. In addition, they implement only three abstractions: threads, address spaces, and kernel memory.

L4-IPC is synchronous and reliable. That is, a communication partner is blocked until the respective other partner becomes ready to communicate, until an error occurs, or until a timeout expires; and, both communication partners are informed about errors respectively about the successful transmission of the message. Interrupts, page faults and exceptions are translated into IPC messages to a respective handler thread [10]. Upon a successful rendezvous, L4 copies the specified thread-local registers and the specified capabilities from the sender to the receiver.

[10] Fiasco-OC [WL10] and seL4 [DdEE] support also asynchronous interrupt notifications.

In old L4 versions [Lie96, Lie99, Sch96, EHL98], these registers are limited by the general-purpose registers of the processor that are not used for other parameters. Since L4 Version X.2 [DLSU04], L4 microkernels implement thread-local message registers. These message registers and other virtual registers can be implemented with a thread-local data structure called *user-level thread control block* (UTCB).

A capability is a kernel-protected tuple, which consists of a set of access rights and a reference to an object on which these rights can be executed. In L4, capabilities are transferred in IPC messages (L4-map IPC). L4-map grants the sender the implicit authority to revoke the transferred access rights. Revocation is by means of the L4-unmap system call. L4-unmap revokes transferred access rights recursively from all address spaces whose threads have directly or indirectly received the unmapped capability from a thread in the unmapping thread's address space.

To send messages (and to transfer capabilities), the sending thread must hold a capability in its address space that conveys send authority to a communication-channel kernel object. In L4.Sec [KV05] and in seL4 [DdEE], these channel objects are called endpoints. They allow multiple threads to receive simultaneously. When a sender sends a message to an endpoint, the kernel selects one of these threads to receive the message. In Nova [Ste09a] and in Fiasco-OC [WL10], channel objects are called portals and IPC-gates, respectively. They are bound to precisely one receiver thread. Together with the message, a thread receives a kernel protected token: the *label*. The creator of the invoked communication-channel object stores this token in the kernel object. The intended use of labels is to identify server-implemented objects. In seL4, Nova and Fiasco-OC, calls to a server implicitly create a reply capability, which conveys the authority to send a reply to the calling client.

In recent kernel versions, a thread must provide kernel memory to create threads, address spaces, communication channels, and other kernel objects. However, the kernel interface for user-controlled kernel-memory management [Hae03] differs. L4.Sec and seL4 implement kernel-memory capabilities that, like user-memory capabilities, refer to a region of physical memory. The kernel is supposed to allocate the data structures of the to-be-created kernel object in this region. Fiasco-OC Factories currently implement a quota scheme on a shared pool of kernel memory. However, a refinement of the Fiasco-OC Factory interface allows for L4.Sec and seL4-like placement controls. In L4.Sec, user memory can be transformed into kernel memory, seL4 implements the reverse transformation. In seL4, user memory is created in kernel memory like all other kernel objects.

Although implementing a specific scheduling policy inside the kernel contradicts the design principle that microkernels should only implement mechanisms and no policies, all L4-family microkernels except the version of L4-Pistachio by Jan Stoess [Sto07] implement a scheduler in the kernel. This scheduler is typically a fixed-priority scheduler. However, Fiasco-OC and L4-CX [Pet09] also experiment with an additional proportional-share scheduler. To avoid malicious or erroneous threads form monopolizing a priority level, the fixed-priority schedulers of L4 enforce a periodically-refilled execution budget. Once a thread has exhausted its budget, the scheduler will not select this thread for execution until the budget of this thread is refilled at the beginning of this thread's next period. Hence, L4 schedulers are typically *budget-enforcing fixed-priority schedulers*.

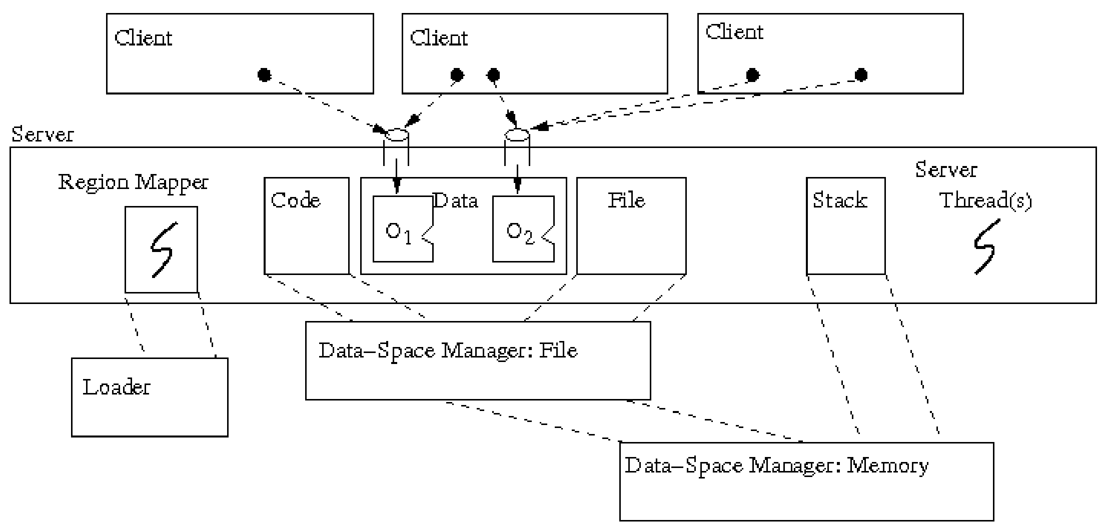

Figure 2.3.: A typical L4 server and its execution environment.

To avoid costly scheduling decisions during the performance critical IPC path, L4-family microkernels implement hand-off scheduling [ABB+86, BALL90, Lie93, LES+97]. With the delivery of the message, the sender of an IPC implicitly donates the remainder of its current timeslice to the receiver. The scheduler is not invoked for this transfer. Often, the invoked server replies well before the next regular scheduling decision. In Fiasco, Wolter et al. [HLR+01] extends hand-off scheduling to a contiguous timeslice donation scheme.

2.5.1. A Typical L4 Server

Application-level servers in L4-family microkernels typically follow a common design. First, these servers run through an initialization phase in which they setup their fault and exception handlers, create the worker threads of this server, and the communication-channel objects for the server objects they implement. Then, they enter a server loop in which they contiguously await requests from clients that they process before they await the next client's request. For the following discussion, it is interesting to investigate these two phases more closely.

2.5.1.1. Server Initialization

When a server starts, its address space contains only the first server thread and a communication-channel capability to this thread's pager. In L4, a *pager* is the thread that receives page-fault messages. The responsibility of the pager is to transfer the appropriate user-memory capabilities to this server. Initially, this is typically a thread in the *loader* that bootstraps this server. Later, it is typically a region-mapper thread [Reu03].

The purpose of the *region mapper* is to translate page faults in valid memory regions into appropriate requests to the server that backs this region. Valid memory regions are for example the server's code and data segments, and regions containing memory-mapped files. The backing servers are the loader, a server for anonymous memory and various file servers.

Memory servers typically implement the data-space interface [ADE+01]. A *data space* is an abstract memory object, a *view* is a section of a data space that can be mapped into the address space of a client. The role of a data-space manager is to back the client region that contains such a view with memory. To do so, a data-space manager may map its own memory or it can rely on the service of other data-space managers. Figure 2.3 illustrates this setup graphically. After creating additional worker threads and the corresponding communication-channel objects for the server-implemented objects, servers typically start executing a server loop.

The representation of user-level server objects with kernel-level communication channels [11] fulfills two purposes:

1. The label of the kernel-level communication channel immediately identifies the data structure of the requested server object; and
2. The transfer and revocation of channel capabilities allows security-policy servers to control whether a thread is able to access the referred server object.

Hence, if the invoked server function can be shown to only access the server object to which the label refers or server objects that are related to this object, access control on server objects can be enforced with the help of the access-control mechanism of the microkernel. In this case, information can flow from a client to such a server object only if the client holds a capability that conveys write access to these objects (e.g., by authorizing a respective server function that writes these objects). Information flows in the reverse direction can happen only to clients that hold a capability which conveys read access.

The purpose of the proposed information-flow analysis is to identify the precise nature of these information flows and to show the server function to access only the expected objects.

In the case of the buffer-cache server (see Section 5.3), communication-channel labels refer to the legitimately-accessible open files. The buffers of the memory pool of the invoking client and the memory-pool meta data are related objects. When a client opens a new file, a new capability is returned for the open file, which establishes this relation.

2.5.1.2. The Server Loop

The C++ pseudo code in Figure 2.4 shows a server loop of a typical server in an L4-based system. Given a message buffer **message**, the server loop invokes the C wrapper function **l4_ipc_wait** until no further IPC errors are reported. IPC errors can report a message 'cut' if a client sends a string message that exceeds the size of the receive window, or a timeout by the client. Servers typically await requests with timeout infinity and send replies with timeout zero. On a reply, IPC errors can indicate that the client has died before the reply could be sent or that the client is not receiving. In these cases, the server typically drops the request and awaits a new one.

The C wrapper functions **l4_ipc_wait** and **l4_ipc_reply_and_wait** are the system-call bindings for two variants of L4-IPC. The first causes the invoker to enter an open receive state in which it awaits messages from all its communication channels, the second invokes the reply capability to send a response to a client and then causes the invoker to enter this open receive state. Both contain assembler code, which loads the system-call parameters into the general-purpose registers of the CPU and causes a kernel entry.

[11] L4.Sec stores the label with the capability, hence, multiple server objects can be represented with a single endpoint.

```
1   Server_Object * label;
2   Message         message;
3    [...]
4
5   error = l4_ipc_wait(label, message);
6
7   do {
8
9     while (error) {
10      // handle IPC error
11       [...]
12
13      error = l4_ipc_wait(label, message);
14    }
15
16    opcode = message.extract_opcode();
17
18    switch(opcode) {
19
20      case f_opcode:
21        label->f(unmarshal_f(message));
22        break;
23
24      case g_opcode:
25        label->g(unmarshal_g(message));
26        break;
27
28      default:
29        message = invalid_opcode;
30    }
31
32    error = l4_ipc_reply_and_wait(label, message);
33
34  } while (true);
```

Figure 2.4.: Server Loop

Per convention, all messages contain the opcode of the invoked operation at a fixed position. The code in Line 16 extracts this opcode. The switch statement in Line 18 checks whether this opcode is valid for the referenced object. If it is not valid, the default case returns an error message to indicate to the client that it has invoked the server with an invalid opcode.

The statements in Line 21 and in Line 25 unmarshal the message parameters for the invoked server functionality. After that, they call a C++ function, which implements the invoked functionality on the server object that is referenced to by **label**. An implicit assumption is here that all communication channels store labels that refer to valid server objects. The **label** is of a derived type of **class Server_Object**.

When the C++ function, which implements the invoked functionality, returns, the server replies to the invoking client and awaits the next request with **l4_ipc_reply_and_wait**.

2.5.1.3. Similarities between Server Loops and L4 System Calls

The implementation of system calls in L4-family microkernels show many similarities to the above server loop: capabilities store the kernel-protected pointer that refers to a kernel object; system-call parameters are passed in the general-purpose registers of the CPU respectively in the invoking thread's UTCB and, in recent kernel versions, an opcode identifies the invoked system call. The two primary differences are:

- The absence of worker threads, and
- The retrieval of information that is stored in the capabilities.

When a thread invokes a system call, the kernel executes this system call on behalf of the invoking thread. For that, it can use the resources (kernel stack, thread control block, etc.) of the invoking thread. A separate kernel thread is not required.

To access the information that is stored in a capability, the kernel has to lookup the capability tables, a data structure that, like the processor page tables, maps address-space local identifies to capabilities. Parameters are either located in the general-purpose registers or in the special-purpose registers of the processor or in the UTCB of the invoking thread. For example, if an IA32 kernel receives a page-fault exception, the page-fault address is passed in the special purpose register CR2 [Cor09, § 2.5 - Vol. 3a].

For the access-control mechanism to control information flows through system calls, the kernel must guarantee that it will only access the kernel object to which the invoked capability refers or an object that is related to this object. An example of a related object is the thread that receives messages that are sent to one of its IPC gates. The role of the information flow analysis of system calls is to establish precisely this guarantee for the system calls of L4-family microkernels.

2.5.2. Confinement

In systems such as L4, where every application-level thread can propagate access rights, it is interesting to know where access rights can propagate and what de-facto access a thread may obtain with the help of other threads [BS79]. The corresponding property is called *confinement*.

A *compartment* is a subsystem that is treated as a single subject by the security policy. A compartment is *confined* [Lam73] if no thread of this compartment can leak information to entities outside this compartment. That is, the information-flow policy must have explicitly authorized all information flows to outside entities such as multilevel servers or the microkernel.

Shapiro [Sha00], Elkaduwe [EKE08] and Boyton [Boy09] show a weaker property: no thread of an *access-confined* compartment can obtain a permission that authorizes a write to an entity outside this compartment unless this permission is received over an explicitly authorized channel.

The information flows of permitted operations and hence the operations through which entities can write to compartment-external entities are assumed axiomatically. For example in Shapiro et al. [Sha00], the functions **reads_from** and **writes_to** formalize the assumed information flows of the system calls of the EROS capability system [Sha99]. When instantiated with the universal lattice for shared-memory programs, the security type system for *Toy* identifies these information flows for the checked system calls and for the checked server invocations.

For a compartment to be access confined in both L4 and in EROS, it must have been started by a trusted loader — the *constructor* [Sha00]. The purpose of this loader is to start compartments and to define the initially authorized channels. It is trusted not to propagate additional (unauthorized) capabilities to the compartments it starts.

2.6. Non-interference-Secure Scheduling

Since their first identification by Schaefer et al. in the context of the KVM/370 security kernel [SGLS77], several solutions have been proposed to eliminate scheduling-related covert channels. Besides fuzzy time [Hu91] and time-partitioning schedulers [Kop98], which I have already discussed in Section 1.3.2, there are two principal approaches to avoid these channels:

1. Information-flow secure schedulers; and
2. Language-based information-flow analyses for applications that run on top of specific classes of schedulers.

To further reduce the remaining covert-channel bandwidth of fuzzy-time systems, Trostle [Tro93] proposes a combination of fuzzy time with channel-bandwidth reducing schedulers.

Hu's lattice scheduler [Hu92] is one such scheduler. Whenever a thread blocks, the lattice scheduler selects the quantum of a ready thread with dominating secrecy level. It then runs this thread unless the quantum is exhausted. Only if the scheduler finds no more ready threads with dominating secrecy level, it resumes the execution of lower-classified threads. In [Tro93], Trostle proposes to further delay this point by idling for a randomly-chosen amount of time.

Both scheduler versions do not eliminate scheduling-related covert channels entirely. They merely reduce their bandwidth. Moreover, many real-time threads (e.g., real-time device drivers) run periodically for short amounts of time and without requiring plain-text access to confidential data. For these threads, the minimal period length, which is achievable with these scheduler versions, is as large as the sum of all quanta. Even if we would modify the lattice scheduler to select the highest-prioritized thread in situations when no more higher-classified threads are ready, period lengths are still at least as large as the sum of the quanta of higher-classified threads. As a consequence, they cannot be used for real-time systems such as our envisaged open microkernel-based system.

In the context of Secure Alpha, Boucher et al. [BCG+94] propose a scheduler that trades real-time performance against covert-channel bandwidth. For that, the scheduler dynamically monitors the bandwidth of covert channels. If the accumulated bandwidth exceeds a certain threshold, the scheduler switches from a real-time scheduling scheme to a scheduling scheme similar to that of the lattice scheduler.

To obtain this threshold, the scheduler considers the time-value functions [Jen92] of its threads. These functions indicate the importance of running the corresponding threads at a certain point in time. In other words, the time-value function of a thread say whether it is still feasible to delay the execution of this thread to reduce covert-channel bandwidth.

In contrast to our scheduler, Boucher requires a completely new *admission test* to determine whether a given real-time workload will meet its timing requirements. In particular, to guarantee both the in-time completion of all threads and an upper bound on the amount of leaked information, covert-channel bandwidths must be predicted at the time of the admission test. For the budget-enforcing fixed-priority scheduler, which I shall introduce in Chapter 3, a large class of existing admission tests can be reused to determine whether all threads will meet their timing requirements. Like Hu and Trostle, Boucher's scheduler cannot completely avoid leakage over scheduling-related timing channels.

Security-type-system-based approaches complement the discussed OS-level solutions. Volpano and Smith [SV98] and Sabelfeld et al. [SS00] propose static information-flow analyses for programs that execute on top of a uniform, a probabilistic, or an arbitrary scheduler. However, for most practical purposes, these analyses are too restrictive.

Russo et al. [RS06] lifts some of these restrictions by allowing threads to inform the underlying scheduler when only equally-classified threads should run. However, Russo's scheduler cannot prevent external timing leaks.

In the envisaged open microkernel-based system, we seek to run also those programs that cannot be checked by contemporary static information-flow analyses. Therefore, we have to reject solutions that are solely based on static information-flow analyses to prevent leakage. Such a solution may however complement OS-level solutions.

2.7. Prototype Verification System (PVS)

I have machine checked the non-interference proof for the budget-enforcing fixed-priority scheduler in Chapter 3 and the soundness proof for the security type system for *Toy* with the help of an interactive theorem prover: the Prototype Verification System (PVS) [ORS92]. In the following, I introduce the syntax and semantics of the specification language of this theorem prover to the degree it is required for this thesis. I assume the reader is familiar with simple set theory [12].

The specification language of PVS is based on a simply-typed higher-order logic enriched with predicate subtypes, dependent record types, abstract data types, inductive and co-inductive types, and various other features.

PVS provides predefined types for the common data types of programming languages. These include, for example, the natural number type **nat**, the integer type **int** and the boolean type **bool** with the values **true** and **false**.

In addition, PVS allows for the creation of record types and it supports (recursive) abstract datatypes. The following code snippet defines the record type **Pair**, a constant **p** of this type and a variable **q** of this type. The type **Pair** contains two members of type **nat**: **x** and **y**.

```
Pair : Type = [#
  x : nat,
  y : nat
#]

p : Pair
q : Var Pair
```

The member access **p‘x** returns the value of the member **x** of the record constant **p**. A partial update of the member **y** of **p** is written as **p With [(y) := n]**. This update returns a new instance of the record **p** whose member **y** equals to **n** and whose member **x** equals to **p‘x**.

[12] A brief summary of simple set theory can be found in [Wik].

Abstract data types define disjoint unions of tagged variants. PVS allows abstract data types to be simple recursive. The following example defines the recursive abstract data type **List**.

```
List [T : Type] : Datatype
Begin
   null : null ?
   cons(car : T, cdr : List ) : cons?
End List
```

The type **T** is a parameter of this abstract data type. It denotes the type of list elements. PVS allows abstract data types and theories to be parametric.

The type **List** contains two variants: the constructor **null** for the empty list and the constructor **cons** for the list that starts with the head-element **car** and whose tail is the list **cdr**. The constructor **cons** is recursive because it takes a list as its second parameter.

The identifiers **car** and **cdr** are accessors, that is, partial functions from List to the types of the respective parameter. **car(l)** returns the head element of the list **l**, **cdr(l)** returns the tail of **l**. For example **car(cons(e, null)) = e**. The type checker prevents **car(null)** and **cdr(null)** because both are only defined for the non-empty list.

The predicates **null?** and **cons?** are recognizer predicates for the corresponding variants. For example, **null?(l)** returns true if and only if **l** is the empty list **null**. PVS uses the same syntax to denote partial updates of abstract data types and of record types.

Abstract data types come with an induction scheme called: *structural induction*. According to this scheme, a predicate **P** holds for all members of an abstract data type **A** if

1. **P** holds for all members produced by non-recursive constructors (the base case), and

2. **P** can be concluded for all recursive constructors from the precondition that **P** holds for the parameters of these constructors that have type **A**.

The keyword **Type** defines a new type. For example, the following code snippet defines two new types **X** and **Y**.

```
X : Type
Y : Type = (pred?[T])
```

The type **X** is not further specified. **Y** contains all elements of the type **T** for which the predicate **pred?** holds. Hence, **Y** is a predicate subtype of **T**. The notation **(pred?[T])** is syntactic sugar for $\{$**y : T** $|$ **pred?(y)**$\}$.

In addition to structures (i.e., records) and tagged unions (i.e., abstract datatypes), PVS also supports functions as first class types. The notation

```
fn( x : X )( y : Y ) : Recursive bool = ...
  Measure ... By ...
```

defines a recursive function **fn** of type **bool** with two parameters of type **X** and **Y**, respectively. Actually, this notation stands also for a function from elements of type **X** to functions of type **[Y** $\rightarrow$ **bool]**. The keyword **Recursive** denotes a recursive specification of **fn**. To ensure that the function is total (i.e., defined for every value of its domain), PVS requires a well-founded order on the parameters of this function. The parameters for this order have to be provided after the keyword **Measure**, the order after the keyword **By**. Lambda notation allows for an inline definition of functions. For example, λ **(x : X)(y : Y) : true** is a function of the same type as **fn**.

PVS collects specifications and lemmas in theories. The proofs of these lemmas are kept in separate files. Like abstract data types, a **Theory** can be parametric. To use the definitions and lemmas of one theory in another theory, the former must be imported with the keyword **Importing**.

Lemmas have the form **name : Lemma spec** where name is a theory-local name for this lemma and spec is the specification. In PVS, the common mathematical constructs are available for specifications: $\Longrightarrow$, $\wedge$, $\vee$, $\Longleftrightarrow$, **Exists (t : T) : ...** and **Forall (t : T) : ...** . In addition, PVS provides a conditional statement: **If ... Then ... Else ... Endif** with the expected semantics and a selection statement for the variants of abstract data types. For example,

```
Cases list_var Of
  cons(h, t) : ...
  null       : ...
EndCases
```

evaluates to the expression on the respective right-hand side of the line matching the variant of **list_var**. The variable **list_var** is of type **List[T]**.

PVS allows the construction of predicate subtypes from arbitrary predicates. Hence, typechecking in PVS is undecidable. Whenever PVS cannot automatically deduce the correct type of a statement, it generates a proof obligation called *type correctness constraint* (TCC). To avoid vacuous results, all TCCs have to be proven in the prover component of PVS.

Proofs in PVS are developed interactively by applying proof commands to the individual goals of a proof. There are proof commands for the standard simplification and verification techniques such as induction, if-lifting and the simplification of binary decision diagrams (BDDs). In addition, PVS provides proof commands for the application of previously shown lemmas.

The prover component of PVS maintains for each proof a *proof tree*. The nodes of this tree denote the *proof goals*. Leaf nodes stand for open proof goals. Each proof goal is represented as a sequence of *antecedents* $(A_1, ..., A_n)$ and *consequents* $(B_1, ..., B_m)$. With the help of the proof commands, the user is expected to show that $A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$ holds. In the interactive proof mode, PVS uses $\vdash$ instead of $\Rightarrow$ and presents antecedents in the lines above $\vdash$ and consequents in the lines below this mark (see the proof of **append_null** below).

To give an idea how a PVS proof looks like, let me repeat the proof of **append_null** from the PVS prelude. It shows that appending the empty list to a list **l** results in precisely this list. The function **append** is recursively defined as:

```
append(l, tail ) : Recursive List[T] =
  Cases l Of
    null : tail ,
    cons(h,t) : cons(h,append(t,tail ))
  EndCases
 Measure length(l)
```

The specification of **append_null** is:

```
append_null : Lemma
  Forall (l : List [T]) : append(l, null ) = l
```

The proof of this lemma uses four proof commands:

- **(induct l)** invokes the structural induction scheme of the list l,
- **(skolem *)** replaces all universally-quantified variables with arbitrary but fixed values of this type,
- **(expand "append")** replaces the function **append** with its definition, and
- **(replace -1 1)** replaces in the consequent {1} all occurrences of the left-hand side of the equation in the antecedent {−1} with the right-hand side of this equation.

The proof of **append_null** proceeds as follows. Initially, the proof tree contains one goal at the root node, which contains the specification of the lemma to be shown:

```
  |-------
{1}   Forall (l : List[T]) : append(l, null)
```

Structural induction over **l** and skolemization **(skolem *)** of the universally-quantified variables spawns two new proof goals as children of this root node:

```
  |-------
{1}   append(null, null)
```

and

```
{-1} append(cons2_var!1, null) = cons2_var!1
  |-------
{1}   append(cons(cons1_var!1, cons2_var!1), null) =
     cons(cons1_var!1, cons2_var!1)
```

The proof command **(expand "append")** solves the first goal because |− **null = null** holds trivially. The same command applied to the consequent {1} simplifies the second goal to

```
{-1} append(cons2_var!1, null) = cons2_var!1
  |-------
{1}   cons(cons1_var!1, append(cons2_var!1, null)) =
       cons(cons1_var!1, cons2_var!1)
```

This goal holds trivially after **(replace −1 1)** replaces **append(cons2_var!1, null)** in the consequent of this goal with **cons2_var!1**, the right-hand side of the antecedent {−1}.

In practice, PVS proofs tend to become rather large. Also, little insight can be obtained from the commands and from the order in which they are applied. This is in particular the case if proof commands such as **grind** are used, which combine several simplification steps in one command. In this thesis, I will therefore refrain from presenting the detailed PVS proofs. Instead, I give an informal direction how the proofs work and refer the interested reader to the published sources [Völ10, Völ08b, VHH08a].

3. Avoiding External Timing Channels in Fixed-Priority Schedulers

This chapter identifies scheduling-related timing channels in fixed-priority schedulers and presents a budget-enforcing fixed-priority scheduler that provably eliminates these channels.

Fixed-priority schedulers always execute one of the highest prioritized ready threads. If more than one such thread exists, a second scheduling policy determines which of these highest prioritized ready threads should run. FIFO and Round Robin are the two most prominent examples of policies for equally prioritized threads.

Budget-enforcing fixed-priority schedulers further constrain the threads they run with a periodically refilled execution budget. Budget-enforcing schedulers execute only threads with positive execution budgets. Running threads consume their budgets.

Essentially, the secure scheduler, which I shall introduce in this chapter, works in the same way as a standard budget-enforcing fixed-priority scheduler. The two fundamental differences are the countermeasures it implements to avoid leakage over scheduling-related timing channels. These countermeasures are:

- **Countermeasure 1:** treat possibly leaking threads as if they where ready, and
- **Countermeasure 2:** defer the points in time when possibly leaked-to threads resume their execution.

In Section 3.3, we shall see in greater detail that the first countermeasure prevents leakage due to alterations in the execution and blocking behavior of higher prioritized threads. The second countermeasure prevents leakage caused by non-preemptively executing lower prioritized threads.

Structure of this Chapter

The remainder of this chapter is organized as follows: Section 3.1 introduces *ReThMo*, a non-standard task model to characterize a large class of classic real-time workloads for the purpose of proving non-interference for fixed-priority schedulers. Task models are typically designed to describe the parameters of real-time workloads for *admission tests*, that is, for tests that seek to determine whether all threads will complete in time (i.e., before their deadline) respectively whether they will meet their timing requirements. Section 3.1.1 discusses the issues that arise when constructing task models for the purpose of describing scheduler workloads for non-interference proofs. Section 3.1.2 introduces the thread scheduling parameters of *ReThMo*, and Section 3.1.3 demonstrates the expressiveness of the proposed task model by describing how classic real-time workloads map to *ReThMo*.

In Section 3.2, I investigate possibilities to leak information through fixed-priority schedulers. Besides the more obvious channels from higher prioritized threads to lower prioritized threads, we shall see how lower prioritized threads altering their non-preemptive execution behavior can leak to higher prioritized threads.

Section 3.3 introduces the budget-enforcing fixed-priority scheduler and the countermeasures it applies. I discuss variations of this scheduler for transitive and intransitive information-flow policies, for FIFO and Round Robin, and alternatives for budget-consumer threads to deal with treated-as-ready blocked threads.

In Section 3.4, I present the formalization of this scheduler in PVS and its machine-checked non-interference proof.

A discussion of the preserved real-time guarantees (Section 3.5) and of practical matters (Section 3.6) concludes this chapter.

The results, which I present in this chapter, are in part based on joint work with Claude-Joachim Hamann and Hermann Härtig. They are documented in a publication [VHH08b] at the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08).

Notational Conventions

The following notational conventions apply to the remainder of this chapter. I write τ_h and τ_l to denote that τ_h is a thread with a higher or equal priority than the thread τ_l.

Given a set T of threads to schedule, I denote with the set $T_{low}(\tau)$ the subset of T that contains all threads with a lower or the same priority than τ. $T_{high}(\tau)$ is defined accordingly as the set of higher or equally prioritized threads.

As introduced in Section 2.3, I denote information-flow policies as triples: $(L, \leq, dom)$. Such a triple consists of a set of secrecy levels L, the dominates relation $\leq$ and the domain dom. Here, dom assigns each thread its secrecy level. I write τ_H and τ_L to denote that τ_H is higher or equally classified (i.e., $dom(\tau_L) \leq dom(\tau_H)$).

In the non-interference proof in Section 3.4, I shall not require $(L, \leq)$ to be a lattice. Instead, it suffices that $\leq$ is reflexive and uniquely bounded from above and from below. In particular, $\leq$ needs not to be transitive. For the arguments about intransitive information-flow policies, recall the definition of an *intransitive pass* as the triple of secrecy levels (s, m, r) with $s \leq m \wedge m \leq r \wedge s \not\leq r$ and the definition of an *intransitive point* as the secrecy level m in the middle of such a pass (see Section 2.2.2).

3.1. The *ReThMo* Task Model

This section introduces *ReThMo*, a task model to characterize the workloads of budget-enforcing fixed-priority schedulers for the purpose of proving these schedulers non-interference secure.

3.1.1. Task Models for Non-interference Proofs

Task models define the parameters that characterize the behavior of real-time and best-effort workloads and of the individual threads [1] of these workloads.

Standard task models, such as the periodic task model [Liu00, Chapter 3.3], are primarily designed to characterize threads for offline admission tests. For these tests to work, the values of thread parameters must already be known while the system is still offline. Therefore,

[1] In the literature, the term *task* is used both for the set of jobs that jointly provide some functionality and for the set of threads that share the same address space. In this thesis, I will call the set of jobs a *thread* to avoid confusions, which arise from this ambiguity.

task models often describe threads with a-priori known parameters that approximate their real behavior in a way that is safe for the admission.

An approximation of thread behaviors is safe for the admission if the real-time guarantees of admitted threads are preserved. Two prominent examples of such approximated parameters are the worst-case execution time (WCET) of a thread and the *critical instant* as an approximation for arbitrary release times. The WCET is an upper bound on the actual time a thread executes. The *critical instant* is the combination of job release times that leads to the worst-case response times of the jobs [Liu00, Chapter 6.5.1] of a thread. The *response time* of a job is the time between its release and the instant when it completes. WCET and critical instant are safe approximations because threads that are admitted with these parameters will complete in time even if they execute shorter than their WCET and even if the jobs of this thread are released at different points in time.

However, for proving a scheduler non-interference secure, task models that are based on approximated parameters risk overlooking information flows due to variations in the actual behavior of a thread. For an information-flow analysis, critical-instant analyses are not sufficient because an analysis of the critical instant for a thread says nothing about the information flows of threads with earlier or later released jobs. The same argument holds for execution time approximations with WCETs.

Fortunately, for non-interference proofs, the precise values of thread scheduling parameters need not to be known a-priori. In *ReThMo*, I shall therefore use parameters, which describe the behavior of threads precisely but whose values are typically not known until the thread reacts in the described way. In the non-interference proof one can easily deal with such unspecified parameters by assuming them to be arbitrary but fixed. However, when modelling a scheduler, one must take care not to rely on the values of parameters at a time when these cannot be known.

For example, *ReThMo* denotes the execution and blocking behavior of a thread as an action trace, which describes the intention of the corresponding thread to do some work or to yield the CPU to other threads if the scheduler would select it at a certain point in time. Obviously, the scheduling decision for such a point in time must not depend on the future actions of a thread because these cannot yet be known. I shall therefore require *ReThMo*-based schedulers to be well formed:

Definition 9. Well-formed scheduler

A scheduler, which is based on the ReThMo *task model (see below) is* well formed *if and only if any scheduling decision that it makes for some point in time t depends only on parameters that are already known at this point in time. For explicitly timed parameters (such as time-to-value mappings) the scheduler may only rely on values whose time t' is earlier or equal to t.*

Release points are an exception. Because *ReThMo* abstracts from thread-releasing events, scheduler models have to check whether the next release point of an inactive thread has yet occurred. To do so, the scheduler model must read the next release point value, whose precise value may not yet be known. The only information that is known is that the value must denote some future time if the release point has not yet occurred. For this reason it is safe to compare release points against the point in time for which the current scheduling decision should be made and to use the result of this comparison to denote that the release point has not yet occurred. A *ReThMo*-based scheduler model in which future release points are used in some other way is not well formed.

It is easy to see that the scheduler model for the budget-enforcing fixed-priority scheduler, which I shall introduce in Section 3.4, is well formed: it only accesses the *ReThMo*-parameters in the above described way.

3.1.2. Thread Scheduling Parameters

The *ReThMo* task model is characterized by the following parameters, with which it describes the behavior of threads. In *ReThMo*, a thread τ_i of the set of threads T is characterized by a possibly infinite sequence of jobs. A job is a unit of work that the system executes [Liu00, Chapter 2.1]. Sequences of jobs are not necessarily periodic. I shall write $\tau_{i,k}$ to denote the k^{th} job of the thread τ_i. Unless explicitly stated otherwise, I assume that the scheduler runs all threads in T on the same CPU. The following parameters characterize the job $\tau_{i,k}$ of a thread τ_i ($k \in \{0, 1, \ldots\}$):

release point $r_{i,k}$**:** the absolute point in time at which $\tau_{i,k}$ becomes eligible for execution;

relative deadline $d_{i,k}$**:** the amount of time after $r_{i,k}$ by which $\tau_{i,k}$ must have finished;

execution budget $eb_{i,k}$**:** an upper bound on the time $\tau_{i,k}$ is allowed to execute; and

total budget $tb_{i,k}$**:** an upper bound on the time $\tau_{i,k}$ is allowed to either execute or to block.

The scheduler keeps track of the remaining budgets of a job. I denote the remaining execution budget by $eb\_rem_{i,k}$ and the remaining total budget by $tb\_rem_{i,k}$. The following four parameters are common to all jobs of a thread:

priority $prio_i$**:** the fixed priority [2]. From all ready jobs, a fixed-priority scheduler chooses between the ones with the highest priority. The precise choice depends on the scheduling policy for equally prioritized jobs (e.g., FIFO or Round Robin);

maximum delay max_delay_i**:** an upper bound on the contiguous time jobs of the thread τ_i may execute non-preemptively; and

action trace $actions_i$**:** a not further specified trace of the actions that the jobs of the thread τ_i will perform.

A released job may perform one of the following **actions**: it may sleep for some while, it may wait for some resource, it may wait for the arrival of a message from another thread or for the occurrence of an external event. In all these cases, I say the job *blocks*. In addition, a job may choose to *execute preemptively* or to *execute non-preemptively*. A job that has finished its work can *stop*. In this case, the thread of this job will continue with the next job at the release point of this next job.

Depending on these actions, on the actions of the jobs of other threads, and on the decisions of the scheduler, a job is in one of the following states.

Running: the job is released and holds all resources it requires, the processor included. The scheduler has selected this job for execution and the job executes preemptively.

[2] I shall also write $prio(\tau_i)$ for the priority $prio_i$ of the thread τ_i.

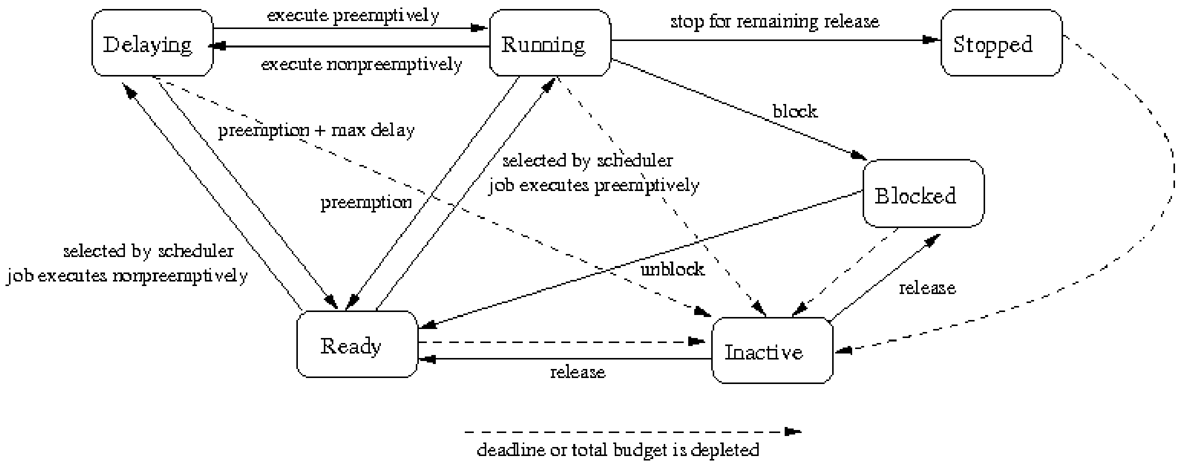

Figure 3.1.: Thread states and their transitions.

Delaying: this state is identical to running except that the job executes non-preemptively. Thereby, it delays the points in time when higher prioritized threads are able to preempt this thread. The scheduler bounds the time by max_delay_i that a job $\tau_{i,k}$ can continue to stay in this state after a preemption has occurred. After this time, the scheduler forcibly preempts $\tau_{i,k}$.

Ready: the job is released and holds all resources with the exception of the processor. For the job to become running or delaying, the scheduler must select it.

Blocked: the job is released but blocks (e.g., because it waits for some resource or for some external event). A job releases the processor when it blocks to allow other ready jobs to run. *ReThMo* does not require a job to run prior to blocking. For example, if a job has requested a resource that is not available until the next job of this thread is released, this next job is released as blocked. Hence, the transition from inactive to blocked.

Stopped: a job that has finished its work can stop. In this case, the thread awaits the release of its next job. A job that has exhausted its execution budget (though not necessarily its total budget) stops automatically. Blocked and stopped jobs continue to consume their remaining total budget. After that, they become inactive.

Inactive: jobs that have exhausted their total budgets (though not necessarily their execution budgets) and jobs whose deadlines have passed are inactive. The thread of an inactive job awaits the release of its next job. Without loss of generality, I assume that a job of a thread becomes inactive before the next job of this thread is released.

In addition, I say:

Active: a job is active if it is not inactive.

An active job can be running, delaying, ready, blocked or stopped. Figure 3.1 presents the transition diagram for these states. The total budget of an active job is not depleted and its

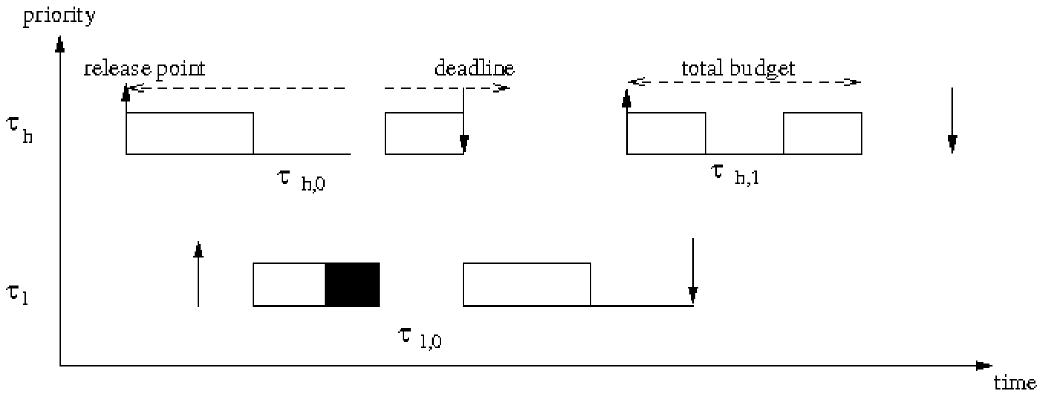

Figure 3.2.: Illustration of the *ReThMo* thread-scheduling parameters. τ_h runs at a higher priority than τ_l.

deadline has not passed yet. In the following, I shall say a thread is in a certain thread state if its current job is in this state. I say a thread runs if its current job either executes preemptively or non-preemptively.

At a first glance, *stopped* and *inactive* seem to express the same state. This is not the case: a job that has stopped still possesses a positive remaining total budget and its deadline has not yet passed. I introduce the distinction between stopped and inactive jobs here because later in Section 3.3, I have to distinguish jobs that stopped voluntarily from jobs that were forcibly deactivated by a passing deadline or as a result of a depleted total budget [3].

Figure 3.2 illustrates the *ReThMo* thread scheduling parameters in an example schedule with two threads: τ_h and τ_l. It shows the execution of two jobs of the thread τ_h and of one job of the lower prioritized thread τ_l. Release points are denoted by upward arrows. Absolute deadlines (i.e., $d\_abs_{i,k} = r_{i,k} + d_{i,k}$) are denoted by downward arrows. The first job of τ_h starts executing and then blocks. At this point in time, the scheduler selects the current job of the next lower prioritized ready thread: the first and only job $\tau_{l,0}$ of τ_l. At first, $\tau_{l,0}$ executes preemptively (white bar). Then, it starts executing non-preemptively (filled bar). Although the first job $\tau_{h,0}$ of τ_h unblocks (end of thin line), the scheduler continues to run $\tau_{l,0}$ until either $\tau_{l,0}$ resumes executing preemptively or until $\tau_{l,0}$ has executed non-preemptively longer than max_delay_l. In both cases, $\tau_{h,0}$ resumes its execution until the scheduler deactivates this job at its deadline. The second job $\tau_{h,1}$ of τ_h executes and blocks longer than $tb_{h,1}$. Once it has consumed this total budget, the scheduler deactivates $\tau_{h,1}$. At this time, the absolute deadline $d\_abs_{h,1} = r_{h,1} + d_{h,1}$ is still in the future.

Note that the total budget of $\tau_{h,0}$ is larger than the total budget of $\tau_{h,1}$. *ReThMo* explicitly allows differing parameter values for the individual jobs of the same thread. This way and

[3] Note that transitioning a blocked (or stopped) thread to invalid does not necessarily induce scheduling overhead. For example, if a concrete implementation of a *ReThMo*-based scheduler stores the absolute point in time t when such a thread has released the CPU, this timestamp reveals whether the thread is still blocked or whether it is already inactive. The latter is the case if the absolute deadline of this thread (i.e., $r_{i,k} + d_{i,k}$) is in the past or if t is longer than $tb\_rem_{i,k}(t)$ in the past. Here, $tb\_rem_{i,k}(t)$ is the remaining total budget at the time t when the thread has released the CPU.

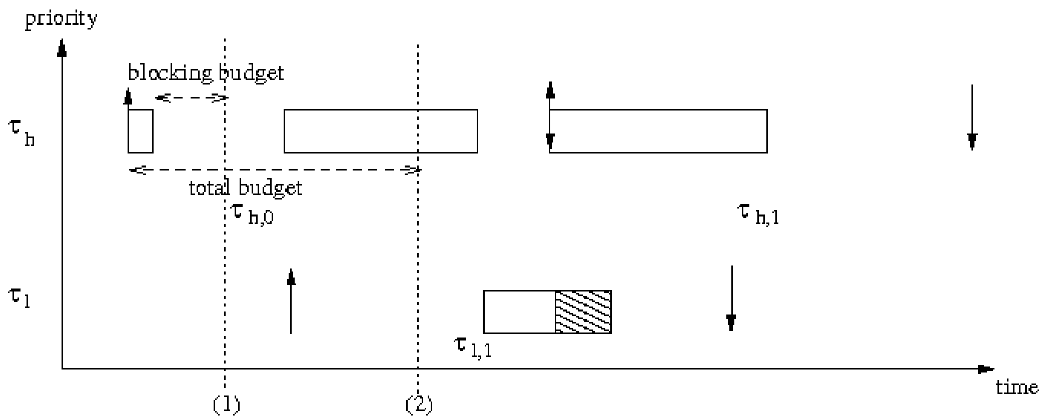

Figure 3.3.: The unconstrained blocking of the higher prioritized thread τ_h causes a deadline miss of the lower prioritized thread τ_l (denoted by the shaded part of τ_l). Schedulers that enforce either blocking budgets or total budgets can avoid these misses.

because the values of most scheduling parameters are taken as arbitrary but fixed, *ReThMo* is able to characterize a large class of existing real-time workloads. In Section 3.1.3, I shall elaborate on the expressiveness of *ReThMo* by giving several examples of common real-time workloads and how they map to the *ReThMo* thread-scheduling parameters.

3.1.2.1. Budget Enforcement

In *ReThMo*, a job is characterized by two budgets: an execution budget and a total budget. In the following section, I motivate this choice and discuss an alternative where jobs are constrained by an additional blocking budget instead of a total budget.

Obviously, for lower prioritized threads τ_l to meet their admitted real-time guarantees, a scheduler must enforce the execution budgets of the jobs of equally and higher prioritized threads τ_h. Otherwise, if such a thread τ_h is malicious or erroneous, τ_l risks missing its deadline.

In addition to unconstrained execution, also unconstrained blocking can cause lower prioritized threads to miss their deadlines. Figure 3.3 depicts such a scenario: the first job $\tau_{h,0}$ of the higher prioritized thread τ_h blocks such that a significant part of its work remains when τ_l's job $\tau_{l,0}$ is released. The time that remains in between $\tau_{h,0}$ finishing its executing and $\tau_{h,1}$ starting its execution does not allow $\tau_{l,0}$ to finish before its deadline. I assume here that τ_l was successfully admitted under the assumption that $\tau_{h,0}$ blocks no longer than the point in time marked as (1) and that some error or malpractice has caused $\tau_{h,0}$ to exceeded its admitted blocking time.

To enforce limited blocking times, two principle approaches are imaginable:

- limit the blocking of a job, or
- limit the total time that a job can either execute or block.

In the first case, the scheduler enforces a blocking budget $bb_{i,k}$. In the second case, it enforces a total (blocking and execution) budget.

In case a scheduler enforces blocking budgets $bb_{i,k}$, it deactivates the corresponding jobs $\tau_{i,k}$ once their accumulated blocking time exceeds their blocking budgets. In the scenario in Figure 3.3, such a scheduler would deactivate $\tau_{h,0}$ at the point in time marked by (1). The remaining execution of $\tau_{h,0}$ is dropped.

In case a scheduler enforces total budgets $tb_{i,k}$, jobs $\tau_{i,k}$ may continue to execute after they have exceeded $bb_{i,k}$. However, the time that they can execute after this excess it reduced accordingly. In total, after their execution or blocking exceeds $tb_{i,k}$, the scheduler deactivates these jobs. In Figure 3.3, the point in time when such a scheduler would deactivate $\tau_{h,0}$ is marked by (2). At this time, τ_l receives sufficient time to complete before the release of $\tau_{h,1}$ and hence before its deadline.

In the following, I shall assume an enforcement of execution and total budgets. Adjusting the presented results for a scheduler that enforces blocking budgets instead of total budgets is straightforward: countermeasure 1 has to last until both the execution budget and the blocking budget of a possibly leaking thread are depleted. That is, the scheduler must treat a possible leaking blocked thread as ready until its remaining blocking budget is depleted and, after that, it must defer the deactivation of this thread until its remaining execution budget is depleted as well. In Section 3.3, we shall return to this point in greater detail.

3.1.3. Expressiveness

ReThMo is sufficiently expressive to describe the real-time workloads of many standard task models. In this section, I demonstrate how strictly periodic threads, sporadic threads and aperiodic threads map to *ReThMo*. In addition, I show how *ReThMo* can describe deferrable servers and the real-time workloads of time-partitioning schedulers. Deferrable servers are a means to schedule aperiodic and sporadic threads together with periodic threads.

ReThMo can also be used to describe the workloads of proportional share schedulers. However, the absolute and relative errors between allocated and received shares increase when threads of these workload are scheduled on budget-enforcing fixed-priority schedulers. Section 3.1.3.5 substantiates this point.

3.1.3.1. Strictly Periodic Threads

A strictly periodic thread τ_i is characterized as usual by a phase Φ_i and by the triple (Π_i, e_i, d_i). The phase determines the release point $r_{i,0} = \Phi_i$ of the first job of τ_i. Subsequent jobs of this thread are released at equidistant points in time (i.e., $r_{i,k+1} - r_{i,k} = \Pi_i$). Hence, the release point of the k^{th} job of τ_i is $r_{i,k} = \Phi_i + k\Pi_i$. The parameter e_i stands for the execution time of τ_i. In admission tests, e_i is often approximated by the maximum of worst case execution times of the jobs of this thread. In case blocking of strictly periodic threads is taken into account, a further parameter x_i bounds the blocking time of the jobs of τ_i from above. The parameter d_i is the relative deadline of the jobs of this thread. That is, each job must have finished latest at $d\_abs_{i,k} = r_{i,k} + d_i$. Because the release of the next job usually deactivates the current job, I will assume that $d_i \leq \Pi_i$.

The mapping of strictly periodic threads to *ReThMo* is straightforward. A strictly periodic thread τ_i can be described as an infinite sequence of jobs $\tau_{i,k}$ with $r_{i,k} = \Phi_i + k\Pi_i$ and $d_{i,k} = d_i$. The execution budgets for all these jobs are set to $eb_{i,k} = e_i$. The total budgets of these jobs are set to $tb_{i,k} = e_i + x_i$. The action trace of τ_i is set to contain the actions that the jobs of τ_i will execute. Its value remains arbitrary but fixed in the sense that τ_i first decides how to proceed at time t before the scheduler evaluates this decision to determine which thread should run at t.

3.1.3.2. Aperiodic and Sporadic Threads

Unlike for strictly periodic threads, the release points of aperiodic and sporadic threads are not known at the time of the admission. In particular, they need not necessarily recur at equidistant points in time. The period of sporadic threads is the minimal distance between the release points of adjacent jobs (i.e., $r_{i,k+1} - r_{i,k} \geq \Pi_i$ holds for all i, k). Aperiodic threads can have arbitrary release points.

The description of threads as infinite sequences of jobs with possibly differing parameter values and the way in which well-formed schedulers evaluate release points [4] allows for a mapping of the precise behavior of aperiodic and sporadic threads to *ReThMo*. In *ReThMo*, the release points of these threads remain arbitrary but fixed values. The intuition is that these release points denote the time when the corresponding job releasing event occurs. For sporadic threads, minimal interrelease times translate into constraints of the form $r_{i,k+1} - r_{i,k} \geq \Pi_i$ with otherwise arbitrary release points. To argue about non-interference, I shall later require that lower or equally prioritized threads can legitimately observe the releasing events of a thread τ_i. The mapping for the remaining thread-scheduling parameters is as described in the previous section for strictly periodic threads.

3.1.3.3. Bandwidth Preserving Servers

Background execution and bandwidth-preserving servers are two principle approaches to integrate sporadic and aperiodic threads into a schedule of otherwise strictly periodic threads.

Background Execution The easiest way to integrate sporadic and aperiodic threads into a schedule with strictly periodic threads is to execute them in the background (i.e., whenever no strictly periodic thread runs). This way, sporadic and aperiodic threads cannot affect the real-time guarantees of strictly periodic threads. However, the response times of these threads is not optimal.

In *ReThMo*, background execution of sporadic and aperiodic threads can be described by assigning these threads priorities that are lower than the priorities of strictly periodic threads. The other parameters of sporadic and aperiodic threads are thereby set as described in Section 3.1.3.2 above.

Alternatively, sporadic and aperiodic threads can be scheduled hierarchically on top of a strictly periodic background thread. Whenever the scheduler chooses to run the background thread, it selects a ready sporadic or aperiodic thread from its aperiodic-thread queue to run. By setting its deadlines and budgets to Π_i (i.e., $d_{i,k} = eb_{i,k} = tb_{i,k} = \Pi_i$), the background thread will be active at any point in time. It may therefore run sporadic and aperiodic threads each time the scheduler runs no higher prioritized strictly periodic thread. To obtain the action trace of the background thread, we have to combine the action traces of the sporadic and aperiodic threads it runs. Because the action trace is an arbitrary but fixed parameter, the combination rule can be as simple as: if at time t, the background thread τ_b decides to run the sporadic or aperiodic thread τ_i, τ_b's action for t is set to the action of τ_i at time t.

[4] See Definition 9 and the discussion that follows this definition on page 47.

Bandwidth-Preserving Servers In contrast to background execution, bandwidth-preserving servers seek to optimize the response times of sporadic and aperiodic threads without deteriorating the real-time performance of strictly periodic threads.

Much like the background thread, a bandwidth-preserving server runs sporadic or aperiodic threads whenever the scheduler selects this server. However, unlike background thread, the execution of sporadic and aperiodic threads is further constrained by a budget. To not confuse it with the execution and total budgets of *ReThMo*, let us call this budget the *aperiodic-thread budget*. Bandwidth-preserving servers are described by two rules:

- A *consumption rule* specifies how running a sporadic or aperiodic thread consumes the aperiodic-thread budget.
- A *replenishment rule* defines when this budget is refilled.

Polling Server Although it is not bandwidth preserving, let us take a look at the polling server as a first example of a server with consumption and replenishment rules for aperiodic-thread budgets. Whenever the scheduler runs the polling server, the server checks the aperiodic-thread queue to determine whether a sporadic or aperiodic thread is ready. If such a thread is present, the server checks whether the remaining aperiodic-thread budget is positive and, if so, it runs the selected thread until this thread finishes or until the remaining aperiodic-thread budget is depleted. In situations where no ready thread is present in the aperiodic-thread queue, the server discards its remaining aperiodic-thread budget and stops to await its next release point.

With the exception of the execution budgets, the *ReThMo* mapping of a polling server τ_p is identical to that of the background thread. The execution budgets $eb_{p,i}$ are set to the aperiodic-thread budget ab_p (i.e., $eb_{p,i} = ab_p$). Because the jobs of the polling server stop whenever they find no ready threads in the aperiodic-thread queue, the total budgets $tb_{p,i}$ can as well be reduced to ab_p. As we shall see in Section 3.3, this reduction allows us to admit threads τ_l at a lower priority than $prio(\tau_p)$ that are not cleared to receive information from the sporadic or aperiodic threads τ_p.

Deferrable Server A deferrable server is identical to a polling server except that it preserves its aperiodic-thread budget until the end of its period. That is, in situations where no aperiodic or sporadic job is ready, it blocks until the next aperiodic or sporadic thread becomes ready. Consequently, we cannot reduce the total budgets $tb_{d,i}$ of the deferrable server τ_d to the aperiodic-thread budget ab_d. Otherwise, the *ReThMo* mapping of polling servers and of deferrable servers are identical.

An immediate consequence of the setting of $tb_{d,i}$ to Π_d is that the non-interference-secure scheduler allows only lower or equally classified threads to be admitted at priorities lower than that of deferrable servers. We shall return to this point in greater detail in Section 3.3.

3.1.3.4. Time-Partitioning Schedulers

A time-partitioning scheduler [ARI, Section 2.3.1] schedules threads in fractions of a periodically recurring major frame of size Π. These non-overlapping fractions are called partition windows. A partition window w_i is characterized by an offset o_i relative to the beginning of the major frame and by a size s_i. In the k^{th} major frame, the scheduler will activate the i^{th} partition window at $k\Pi + o_i$ for the time s_i. If more than one thread is assigned to such a window, a second scheduling policy is required to select one of these threads. The system idles when all threads of a partition window block or when they have stopped.

In principle, it is possible to map time-partitioning schedules to *ReThMo*. Although, from a scheduling-overhead point of view, it is not advisable to do so. Let τ_i be a thread that executes in the i^{th} partition window. Then the parameters of the k^{th} job $\tau_{i,k}$ of τ_i are set to allow this job to run only during the i^{th} partition window of the k^{th} major time frame. To do so, we set $r_{i,k} = k\Pi + o_i$, and both, the budgets $eb_{i,k}$, $tb_{i,k}$ and the relative deadline $d_{i,k}$ to the size of the partition window s_i. The priority of such a thread is a free parameter, which can be chosen arbitrarily.

3.1.3.5. Proportional-Share Schedulers

Proportional-share schedulers [Wal95] seek to minimize the absolute and relative errors between the time a thread τ_i runs in a sliding window of size t and the proportion $prop_i$ of this window size t that τ_i should receive. Often, tickets are used to characterize this proportion. If a thread holds n_i out of N tickets, it should receive the proportion $prop_i = \frac{n_i}{N}$. It holds that $\sum_i prop_i = 1$.

The two most prominent proportional-share schedulers — the randomized *lottery scheduler* [WW94] and the deterministic *stride scheduler* [WW95] — execute the following basic algorithms. Let the *unit of time* u be a small fixed amount of time.

Lottery Scheduler: Every unit of time u, the lottery scheduler randomly picks a ticket and schedules for one unit of time the thread that holds this selected ticket. Assuming that the individual tickets are picked with equal likelihood, a thread τ_i with n_i tickets receives $\frac{n_i}{N}$ of the CPU time on the average.

Stride Scheduler: The stride scheduler computes for each thread τ_i a *stride* s_i as the inverse fraction of held tickets and total tickets (i.e., $s_i = \frac{N}{n_i}$). The *pass* of τ_i is a virtual time index used to determine which thread to run next. Every u milliseconds, the scheduler selects the thread with the smallest pass to run for one unit of time. Initially the pass p_i of τ_i is set to the stride s_i. Whenever the scheduler runs τ_i, it advances its pass by s_i.

Both, the lottery scheduler and the stride scheduler wastes one unit of time if the selected thread blocks. As a consequence, blocked threads consume only a fraction f of their allocated time. To accommodate for these reduced shares, Waldspurger proposes to extend the lottery scheduler and the stride scheduler with transient compensation tickets [Wal95]. Whenever one of these schedulers selects a blocked thread, it repeats the selection procedure until it finds a thread that is ready. At the time when a blocked thread resumes its execution, it temporarily increases the tickets of this thread to $\frac{n_i}{f}$.

Although an elaborative discussion of non-interference-secure proportional-share schedulers is out of the scope of this thesis, I will briefly return to proportional share schedulers in Section 3.3.10 to investigate the information-flow properties of these schedulers for a mapping of proportional-share workloads to the *ReThMo*-based budget-enforcing fixed-priority scheduler. In the following, I shall introduce this mapping.

	absolute error	relative error
lottery (expected errors)	$O(\sqrt{t})$	$O(\sqrt{t})$
stride	$O(\lvert T\rvert)$	≤ 1
fixed priority	$\leq \Pi p_i(1-p_i)$	$O(\frac{n_i n_j}{n_i+n_j})$

Table 3.1.: Absolute and relative errors of the proposed mapping of proportional-share workloads to a budget-enforcing fixed-priority scheduler. $|T|$ denotes the cardinality of the set of threads T, i.e., the number of threads.

Mapping Proportional-Share Workloads to *ReThMo* Let the set T_{prop} contain the threads of a proportional share workload. Each thread τ_i of this workload should receive a proportion $prop_i$ of the CPU time. In *ReThMo*, a mapping of $r_{i,k} = k\Pi$, $d_{i,k} = \Pi$, and $eb_{i,k} = prop_i\ \Pi$ for each job of a thread τ_i, allows τ_i to execute for $eb_{i,k}$ every Π. Therefore, on the average, it receives the proportion $prop_i = \frac{eb_{i,k}}{\Pi}$. The priority of τ_i remains a free parameter. For the time being, let us assume that $tb_{i,k} = \Pi$ holds for all jobs of all threads. This way, lower prioritized threads can consume any time that higher prioritized threads block. I shall refine this choice in Section 3.3.10.

Errors How good is this mapping? Waldspurger [Wal95] defines the *absolute error* between the allocated and received CPU share a thread τ_i receives in a sliding window of size t as the difference between $t\frac{n_i}{N}$ and the time e_i that τ_i did run during this window. The *relative error* between the allocated and received shares of a pair of threads (τ_i and τ_j) is the absolute error in a system that contains only these two threads. This is a system with $t = e_i + e_j$ and $N = n_i + n_j$. Table 3.1 shows these errors for the lottery scheduler, for the stride scheduler and for the above mapping to a budget-enforcing fixed-priority scheduler. The error formulas for the first two are taken from [Wal95]. The latter can be derived as follows.

If proportional-share workloads are mapped to *ReThMo* as described above, the longest consecutive time during which a thread τ_i does not receive any CPU time is $2\Pi - 2\ prop_i\Pi$. This situation occurs if τ_i has a priority, which is lower than the priority of all other threads, and if all these other threads block for $eb_{i,k}$ units of time during the first period of length Π and not at all in the second of the two consecutive periods. In this situation, the absolute error of τ_i is maximal for a sliding window of size $t = 2\Pi - prop_i\Pi$. During this time, τ_i executed for $e_i = \Pi prop_i$ units of time. Hence, τ_i's absolute error is:

$$\begin{array}{cc} t\ prop_i - e_i & = \\ (2\Pi - prop_i\ \Pi)\ prop_i - prop_i\ \Pi & = \\ \Pi\ prop_i\ (1 - prop_i) & \end{array} \tag{3.1}$$

The maximal relative error of τ_i and τ_j is:

$$(2e_j + e_i)\frac{n_i}{n_i + n_j} - e_i \tag{3.2}$$

because for a system with two threads, the absolute error is maximal if $t = 2e_j + e_i$ and if τ_i executed for $e_i = \Pi\frac{n_i}{N}$. Hence, because e_i is proportional to n_i and likewise because e_j is proportional to n_j, Equation 3.2 is proportional to:

$$(2n_j + n_i)\frac{n_i}{n_i + n_j} - n_i = \frac{n_i n_j}{n_i + n_j} \tag{3.3}$$

In comparison with the errors of the lottery scheduler and of the stride scheduler, errors of the above mapping of proportional-share workloads to a *ReThMo*-based fixed-priority scheduler are much larger. For example, the absolute error of the latter is $\Sigma\frac{eb_i}{|T_{prop}|}$ times the absolute error of the stride scheduler. This difference occurs because the fixed-priority scheduler runs a thread until it has depleted its execution budget. The stride scheduler runs a thread only for one unit of time. In the case of the fixed-priority scheduler, lower or equally prioritized threads must therefore wait significantly longer than in a system with a stride scheduler.

3.2. External Timing Channels in Fixed-Priority Schedulers

The following section investigates external timing channels in fixed-priority schedulers. For the time being, let us assume that threads have distinct priorities. We shall return to equally prioritized threads in Section 3.3.8. Let us further assume that threads have access to precise clocks [5].

The deliberate choice of a high-priority thread τ_h to run, to block, or to stop influences when a lower prioritized thread τ_l can run. The scheduler will not select τ_l during those times when τ_h executes preemptively or non-preemptively. If τ_h blocks or stops, the scheduler may select τ_l to run. I call this form of influence: *direct influence* because τ_h's actions directly affect τ_l.

Direct influence constitutes an external covert timing channel. For example, if a high-priority thread τ_h encodes secrets by running at a certain point in time t to send a 0 respectively by blocking at t to send a 1, a lower-prioritized thread τ_l can read this secret by sampling the precise clock to obtain the points in time it did run and by evaluating these points to determine whether it did run at t. More generally, direct influence reveals information about the execution and blocking behavior of the sending high-priority thread if the lower-prioritized receiver τ_l detects a derivation between the points in time when it would run if no higher prioritized thread would block and the points in time it actually did run.

Obviously, if more than one thread has a higher priority than the receiver τ_l or if other threads share τ_l's priority (see Section 3.3.8), external timing channels due to direct influence are noisy. However, Foss et al. [SAF06] show that direct-influence based channels exist — e.g., the "$channel - \Gamma_{RM}$" for RMS — that are positively deducible. That is, irrespective of the noise of higher or intermediary prioritized threads, there are observable influences that reveal the exact message of the sender. RMS stands for the rate-monotonic scheduling algorithm [LL73], which assigns thread priorities inverse proportionally to thread period length.

3.2.1. Indirect Influence

In addition to direct influence, inter-process communication allows threads to also influence other threads indirectly by directly influencing the sender of such a message. If a thread τ_h directly influences a thread τ_s, it affects the points in time when τ_s is able to run and thereby the points in time when τ_s can send messages to a legitimate receiver τ_r. Therefore, by directly

[5] In Section 1.3.1, we have seen that fuzzy clocks deteriorate the real-time capabilities of our envisaged system, which motivates the above assumption.

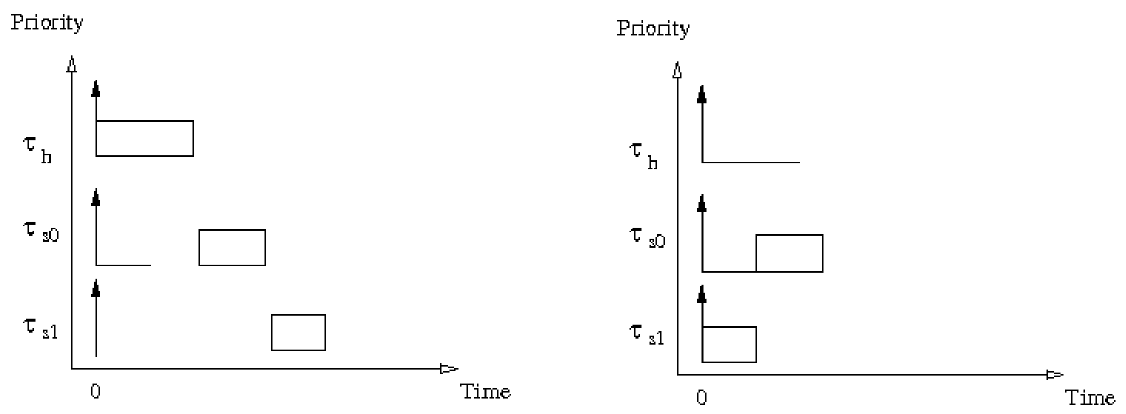

Figure 3.4.: The execution order of τ_{s0} and τ_{s1} and thus the order of their messages depends on the behavior of τ_h. If τ_h executes (left picture) while τ_{s0} blocks, τ_{s0}'s messages arrive at τ_r (not shown) before τ_{s1}'s messages. If τ_h blocks during this time (right picture), τ_{s0}'s and τ_{s1}'s messages arrive at τ_r in the reversed order.

influencing τ_s, a thread τ_h with a higher priority than τ_s is able to convey information to τ_r. I call this form of influence *indirect influence* because τ_h can in general not influence τ_r directly. Instead, it needs to directly influence the sender τ_s. In particular, if τ_r has a higher priority than τ_h, τ_h can influence τ_r only indirectly but not directly. Clearly, a scheduler that avoids all direct influences would also avoid all indirect influences. However, such a scheduler would be overly restrictive. The non-interference-secure scheduler, which I will introduce in greater detail in Section 3.3, will therefore avoid only those direct influences that could cause illegal information flows to directly or indirectly influenced threads. In Section 3.3.4, we shall see that in the case of intransitive information-flow policies the scheduler must avoid also some direct influences where information-flows to the directly influenced thread are legitimate. This is because the directly influenced thread may be cleared to legitimately send messages to other threads to which the influencing thread is not cleared to send information.

Surprisingly, trusted servers cannot avoid covert channels due to indirect influences if their threads are directly influenced. Let us assume that all server threads are trusted not to encode timing information in the messages they send. Then there are still scenarios in which a thread τ_h is able to indirectly influence a thread τ_r that receives messages from these trusted server threads.

Figure 3.4 illustrates such a scenario. The thread τ_h directly influences an intermediate prioritized sender τ_{s0} and a lower prioritized sender τ_{s1} to indirectly influence the receiver τ_r. Assume that all threads are released simultaneously and that τ_{s0} first blocks and then runs for some while. If τ_h runs for the time that τ_{s0} blocks, τ_{s0} runs before τ_{s1}. If τ_h blocks, τ_{s0} runs after τ_{s1}. The order in which the messages of these two threads arrive at τ_r is reversed. As long as communication channels reveal the order in which messages arrive at a thread, the directly influenced threads τ_{s0} and τ_{s1} cannot prevent this external timing channel.

The inability of directly influenced servers to completely eliminate covert channels due to indirect influences has an immediate impact on systems with intransitive information-flow policies:

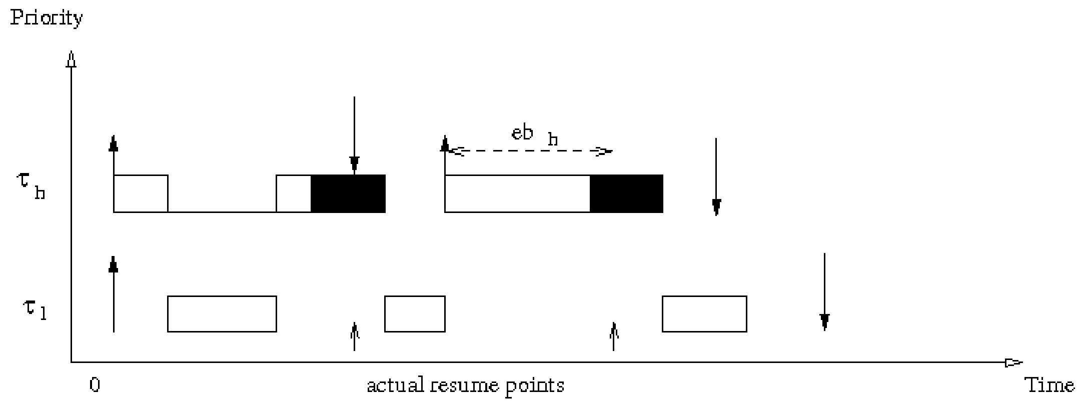

Figure 3.5.: A thread τ_h can influence a lower prioritized thread τ_l by executing non-preemptively when its deadline passes (left) or when one of its budgets depletes (right).

the priorities of server threads that are classified at intransitive points must be sufficiently high to prevent their direct influences. Otherwise, if a sender is able to directly influence a server thread, messages, which the server forwards after sanitizing the transmitted information, could still reveal un-sanitized secrets encoded in the order in which they arrive at the receiver. To prevent a direct influence of a server, all potentially untrustworthy clients, which send to this server, must be lower prioritized than the server threads.

3.2.2. Influence due to Non-preemptive Execution

A third class of external timing channels arises from the non-preemptive execution of lower prioritized threads. In contrast to external timing channels due to direct influence, information flows due to non-preemptive execution are typically directed from lower prioritized to higher prioritized threads.

To turn non-preemptive execution into a covert channel, a low-priority thread τ_l encodes secrets by deliberately choosing between preemptive and non-preemptive execution. In the first case, a higher prioritized thread τ_h is able to preempt τ_l immediately. In the second case, the preemption and therefore the point in time when τ_h resumes its execution is deferred to the point in time when τ_l stops executing non-preemptively. Equally prioritized threads can be influenced in the same way if they can preempt τ_l's execution (see Section 3.3.8). Lower prioritized threads are typically not affected. However, there are two corner-case situations in which a non-preemptively executing thread can also leak information to lower prioritized threads:

1. when a thread executes non-preemptively to exceed a depleted execution budget; and

2. when a thread executes non-preemptively to exceed a passing deadline.

Figure 3.5 illustrates these two corner cases.

Low-level operating-system code executes non-preemptively by disabling all processor interrupts and hence the events that trigger scheduling decisions. *Fully interruptible* operating-system kernels allow interrupts to preempt kernel code at any point in time. However, instead of executing the triggered scheduling decision immediately, they return to the preempted code path if this path has signalled its intent to execute non-preemptively.

Application-level programs can execute non-preemptively by issuing system calls that the kernel executes on their behalf and that contain non-preemptive code paths. Some operating-system kernels even implement mechanisms [MDP96, KWS97] through which applications can defer scheduling decisions. In recent L4-family microkernels [DLSU04, WL10, DdEE, KV05], this mechanism is called *delayed preemption*. A flag in the user-level thread control block of the currently executing thread informs the kernel about its intent to execute non-preemptively. If the kernel interrupt handlers find this flag set, they defer the handling of the interrupt and return control to the preempted application code. However, before doing so, they inform the application about the preemption and program a timer to bound the time that the application program can execute non-preemptively. Once the kernel regains control, either because the application program has voluntarily returned control or because the timer has fired, it processes the pending preemption and executes the triggered scheduling decision.

3.3. A Non-Interference-Secure Scheduler

This section introduces the non-interference-secure budget-enforcing fixed-priority scheduler and the countermeasures it implements to avoid illegal information flows over external timing channels. At first, I give a general overview on the operation of this scheduler. Then, I present the individual parts of this scheduler in greater detail and discuss several variants.

Given a fixed-priority scheduler that enforces total budgets, two practically feasible modifications suffice to eliminate external timing channels. I shall call these modifications *countermeasures* because each of these two modifications addresses a different class of external timing channels.

- **Countermeasure I:** To avoid illegal information flows due to direct and indirect influences, the first countermeasure treats possibly leaking blocked or stopped threads as if they were ready. Modulo preemptions by higher or equally prioritized threads, the scheduler will always run treated-as-ready threads to completion. As a consequence, lower prioritized threads (and some equally prioritized threads) that are not cleared to receive information from such a treated-as-ready thread will not be selected while this first countermeasure is active. Therefore, they cannot observe variations in the execution and blocking behavior of the treated-as-ready thread.

- **Countermeasure II:** To eliminate timing channels due to non-preemptive execution, all scheduling decisions that are triggered by a preemption of a higher or equally prioritized thread are deferred by an amount of time that a possibly leaking non-preemptively executing thread cannot influence. As a consequence, the preempting thread can no longer distinguish between variations in the preemptive and non-preemptive execution behavior of a thread and its deferred resumption due to this second countermeasure.

In Section 3.3.1 and in Section 3.3.5, I discuss the above two countermeasures in greater detail. The scheduler's decision to activate these countermeasures depends on two static predicates, which differ for transitive and for intransitive information-flow policies. In Section 3.3.2, I argue why, although imprecise, predicates have to be static. Section 3.3.3 investigates static predicates for the first countermeasure assuming a transitive information-flow policy. Section 3.3.4 discusses static predicates for intransitive information-flow policies.

Because the first countermeasure treats possibly leaking blocked or stopped threads as ready, a suitable thread must be found to consume the execution and total budget of treated-as-ready threads. For the time being, let us assume that the idle thread plays this role. In Section 3.3.7, I discuss a variant of the scheduler, which allows also other threads to consume the budgets of possibly leaking threads.

Two further assumptions to which we shall stick in the following discussion are that threads have distinct priorities and that jobs have no precedence constraints or other temporal dependencies. I shall lift these restrictions in Section 3.3.8 and in Section 3.6.1, respectively. All threads are assumed to run on the same CPU.

Clearly, if the existence of a thread must not be revealed to lower prioritized threads, the former must not consume any time that would otherwise be available to the latter. At least, a placeholder thread must be visible to this latter thread to reserve time for concealed threads. As a fifth and last assumption I will therefore assume that all threads are cleared to observe the release points, the relative deadlines and the total budgets of higher prioritized threads. In Section 3.6.2, we shall return to the execution of concealed threads behind visible placeholder threads. Section 3.3.10 concludes the discussion of non-interference-secure schedulers with a slight detour to non-interference-secure proportional-share schedulers.

3.3.1. Avoiding Information Leakage due to Direct and Indirect Influences

There are two principle approaches to avoid information leakage due to direct and indirect influence:

1. constrain the execution and blocking behavior of influencing threads, or

2. compensate variations in an influencing thread's execution and blocking behavior to prevent the influenced thread from observing encoded secrets.

Countermeasure I follows the second approach because, as we shall see in greater detail in Section 3.5, it preserves the real-time guarantees of the scheduled threads. *Countermeasure I* is defined as follows:

Definition 10. Countermeasure I

Let $p_{\mathit{influence}}(\tau_h, t)$ be a predicate, which denotes whether τ_h can directly or indirectly influence another thread at the point in time t. The first countermeasure to avoid information leakage due to direct and indirect influence is to treat τ_h as if it is ready whenever $p_{\mathit{influence}}(\tau_h, t)$ evaluates to true. That is, the scheduler selects τ_h whenever no higher prioritized thread is ready or treated-as-ready. If the selected thread τ_h blocks of if it has stopped, the scheduler runs a suitable budget-consumer thread to compensate for τ_h's blocking behavior and to consume τ_h's total budget.

Clearly, for *Countermeasure I* to work, the budget-consumer thread must be ready and it must not send information about τ_h's blocking behavior to lower prioritized threads τ_l that are not cleared to receive information from the influencing thread τ_h. In Section 3.3.7, I will discuss possible choices of suitable budget-consumer threads. For now, let us pick the *idle thread*.

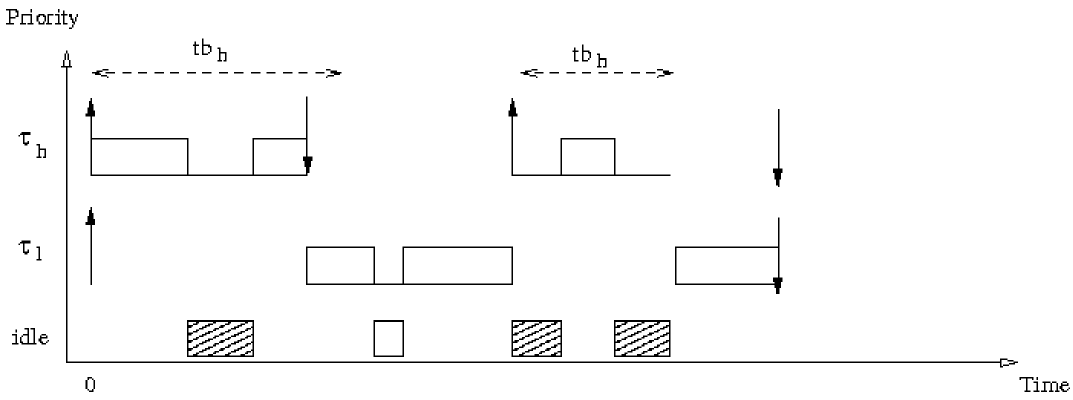

Figure 3.6.: To avoid information leakage due to direct and indirect influences, the modified scheduler prevents τ_l from running when τ_h blocks or when it has stopped. The idle thread consumes τ_h's total budget (shaded bars).

To avoid special-case handling in the scheduler, many operating systems implement a per CPU *idle thread* that is always ready. The idle thread performs no useful work but to idle. In particular, it sends no messages. Therefore, it can safely be classified at the highest secrecy level $\top$ of the information-flow policy, which clears it to receive information from all other threads. Because the idle thread does not send any messages, it will not reveal any information about the points in time during which it runs. Therefore, and because the idle thread is always ready, it can be selected to consume the budget of any other thread in the system. This means the idle thread is a suitable budget-consumer thread for all other threads.

Figure 3.6 shows how *Countermeasure I* avoids information leakage due to direct and indirect influences. Recall from Section 3.2.1 that we can rule out indirect influences by ruling out direct influences of those legitimate receivers that are cleared to send messages to the indirectly influenced threads.

Let us assume that τ_h must not leak any information to τ_l (i.e., $dom(\tau_h) \not\leq dom(\tau_l)$) and hence that $p_{\mathit{influence}}(\tau_h, t)$ holds at least for all points in time t when it is necessary to activate *Countermeasure I*. Whenever τ_h blocks or whenever it has stopped without exhausting the total budget $tb_{h,k}$ of its current job, the scheduler switches to the idle thread to consume this budget. As a consequence, the scheduler prevents τ_l from running until either τ_h's total budget is depleted or until its deadline has passed. Both situations are independent of the actions of τ_h and we assumed τ_l to be cleared to the release and deactivation of the higher prioritized thread τ_h. Therefore, because τ_l cannot distinguish τ_h's execution from the execution of the idle thread to consume τ_h's total budget, τ_h cannot influence the points in time when τ_l runs and hence the observations τ_l can make about τ_h's execution and blocking behavior.

The scheduler avoids information leakage due to indirect influences of a thread τ_r by avoiding direct influences of threads τ_s that are cleared to send to τ_r. Because a thread τ_h, which cannot directly influence τ_s, cannot influence the points in time when τ_s runs, it can also not influence the timing information in the messages τ_s sends. Therefore, a suitable predicate $p_{\mathit{influence}}(\tau_h, t)$, which activates *Countermeasure I* also when such a possible sender τ_s could be influenced directly, avoids also information leakage due to indirect influences.

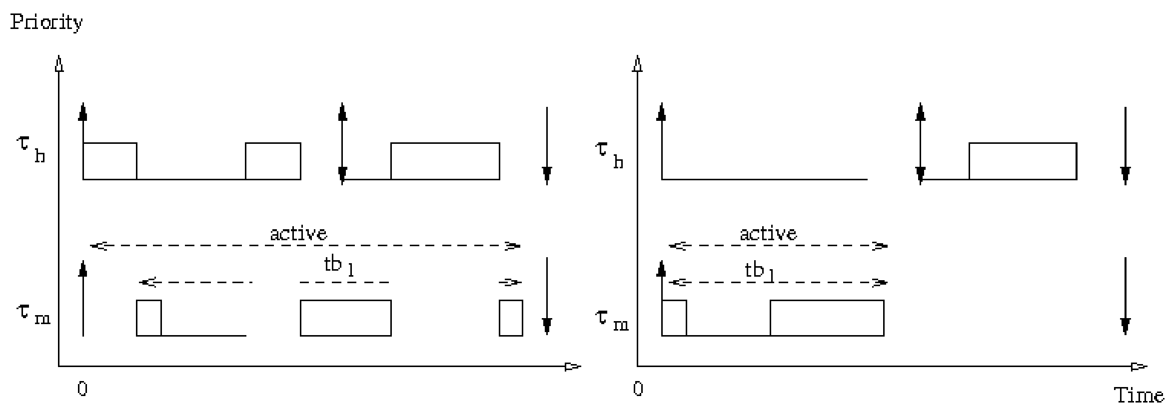

Figure 3.7.: The time that τ_m is able to influence lower prioritized threads depends on the time that higher prioritized threads (e.g., τ_h) execute while τ_m is released.

3.3.2. Suitable Predicates for *Countermeasure I*

In Definition 10, the decision to apply *Countermeasure I* for a thread τ_h at time t depends on the as yet unspecified predicate $p_{\mathit{influence}}(\tau_h, t)$. In the next section, I argue why, although imprecise, $p_{\mathit{influence}}(\tau_h, t)$ must be static. Section 3.3.3 and Section 3.3.4 introduce two candidates — $p_{\mathit{transitive}}$ and $p_{\mathit{intransitive}}$ — for transitive and for intransitive information-flow policies and discuss why they are suitable to avoid information leakage due to direct and indirect influences.

3.3.2.1. Static Predicates are Imprecise

To minimize the scheduling overhead and to allow admission tests to be based on the predicates, which denote when the scheduler has to activate the two countermeasures, these predicates must be static. That is, the decision to treat a possibly leaking thread as ready must not depend on information that is only available when the system is running. For the same reasons, a static predicate is required for the second countermeasure (see Section 3.3.5). However, static predicates are imprecise.

Precise predicates activate a countermeasure only when a thread τ_m can actually leak information by directly influencing another thread. Clearly, this is only the case if no other higher prioritized thread runs and if the influencing thread τ_m is active. Unfortunately, both situations depend on information that is in general only available when the system runs: Whether a higher prioritized thread runs at a point in time t depends on the actions this thread executes and on the execution and blocking behavior of other higher prioritized threads. However, the action a thread will execute at t is typically not known before t (recall Definition 9 on page 47 about well-formed schedulers and time-to-value mappings).

Figure 3.7 demonstrates that thread activation also depends on the release points and actions of higher prioritized threads, that is, on information that is in general not available at admission time. Two threads τ_h and τ_m are shown with two respectively one job $\tau_{h,0}$, $\tau_{h,1}$, and $\tau_{m,0}$. The last job $\tau_{m,0}$ executes and blocks until its total budget $tb_{m,0}$ is exhausted. If the first job $\tau_{h,0}$ of τ_h runs (left picture), τ_m remains active until after τ_h's second job becomes inactive. If τ_h's first job blocks (right picture), τ_m is deactivated due to budget depletion before the second job of τ_h

is released.

To conclude, static countermeasure predicates cannot be precise. They cannot depend on the actions of a thread nor can they depend on the points in time when a thread is active. Conservative predicates, which overestimate the points in time when the respective countermeasure has to be applied are safe as long as the countermeasure is activated at least during all those points in time when a precise predicate would activate this countermeasure.

3.3.3. Transitive Information-Flow Policies

Definition 11 introduces the predicate $p_{transitive}(\tau_h, t)$ for transitive information-flow policies. It overestimates the points in time when the scheduler has to activate *Countermeasure I* to prevent illegal information flows due to direct and indirect influence.

Definition 11. Predicate for Transitive Policies.
The predicate $p_{transitive}(\tau_h, t)$ is a conservative countermeasure predicate for transitive information-flow policies. It is defined as follows:

$$p_{transitive}(\tau_h, t) := \exists\, \tau_l \in T_{low}(\tau_h).\ dom(\tau_h) \not\leq dom(\tau_l)$$

In fact, the result of $p_{transitive}(\tau_h, t)$ does not depend on the parameter t. In the following, I will therefore write $p_{transitive}(\tau_h)$ instead of $p_{transitive}(\tau_h, t)$. The predicate holds for a thread τ_h if and only if there is a lower or equally prioritized thread τ_l that the information-flow policy has not cleared to receive information from τ_h. Remember, the set $T_{low}(\tau_h)$ contains all threads τ with $prio(\tau) \leq prio(\tau_h)$.

The following observations give an intuition why *Countermeasure I* with $p_{influence} = p_{transitive}$ avoids information leakage due to direct and indirect influence in systems with a transitive information-flow policy. In Section 3.4, I substantiate this informal argument with a machine-checked non-interference proof of the budget-enforcing fixed-priority scheduler.

Assume τ_h is a high-priority thread that must not send to a lower prioritized thread τ_l. Then, because $dom(\tau_h) \not\leq dom(\tau_l)$ and because $\tau_l \in T_{low}(\tau_h)$, $p_{transitive}(\tau_h)$ holds for all points in time t. As a consequence, whenever no higher than τ_h prioritized thread is ready or treated-as-ready, the scheduler will select τ_h as long as τ_h is active. During this time, the scheduler will either run τ_h or the idle thread to consume τ_h's total budget; the lower prioritized thread τ_l is not run. Therefore, variations in τ_h's execution or blocking behavior have no effect on τ_l.

A first intuition why *Countermeasure I* with $p_{influence} = p_{transitive}$ avoids also leakage due to indirect influence gives the following case analysis of the values of $p_{transitive}$:

Case $p_{transitive}(\tau_h)$: If $p_{transitive}(\tau_h)$ holds then τ_h cannot directly influence lower prioritized threads $\tau_s \in T_{low}(\tau_h)$. Any timing information that τ_s's messages may carry to a thread τ_r, which τ_h intents to indirectly influence, must therefore be independent of τ_h's actions.

Case $\neg p_{transitive}(\tau_h)$: If the predicate $p_{transitive}(\tau_h)$ does not hold, we have to assume pessimistically that τ_h directly influences a lower prioritized thread τ_s. Through this direct influence, τ_h can affect the timing information in τ_s's messages. It may therefore indirectly influence those threads τ_r to which τ_s is authorized to send (i.e., for which $dom(\tau_s) \leq dom(\tau_r)$ holds).

From $\neg p_{transitive}(\tau_h)$, we know that $dom(\tau_h) \leq dom(\tau_s)$. Because $dom(\tau_s) \leq dom(\tau_r)$ holds and because $\leq$ is transitive, it follows that $dom(\tau_h) \leq dom(\tau_r)$. Hence, if

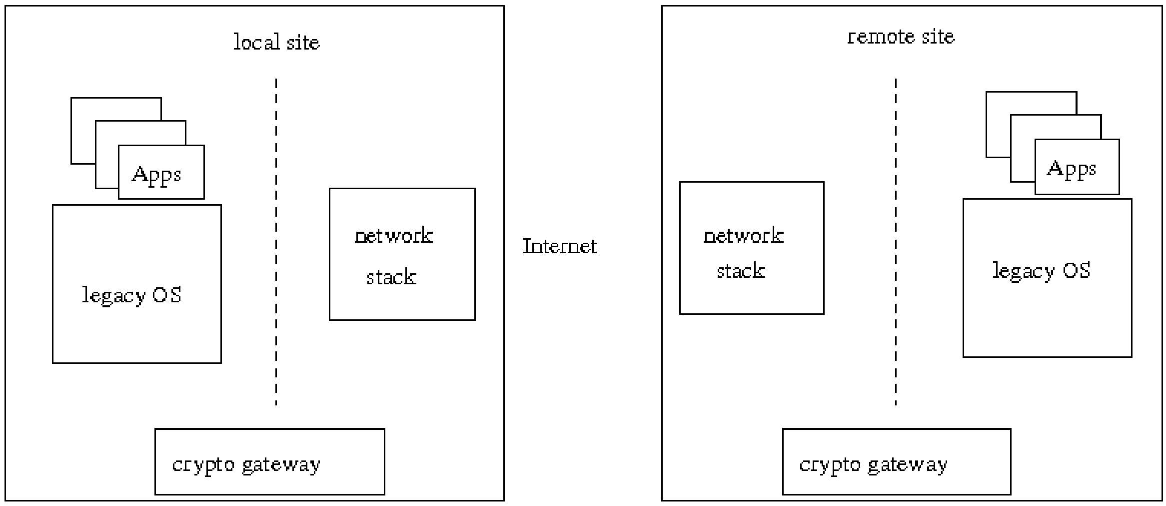

Figure 3.8.: The Mikro-SINA cryptographic gateway. A cryptographic gateway connects a potentially untrustworthy subsystem over the Internet to another potentially untrustworthy subsystem. The shown implementation reuses not necessarily trustworthy network protocol stacks.

$\neg p_{transitive}(\tau_h)$, τ_h is authorized to send to all threads τ_r to which lower prioritized threads (e.g., τ_s) are authorized to send. No leakage can occur.

This concludes the informal argument. For transitive information-flow policies, we have seen that the *Countermeasure I* with $p_{influence} = p_{transitive}$ avoids leakage due to direct and indirect influences. In the next section, we shall see that $p_{transitive}$ is not sufficient for intransitive information-flow policies. Therefore, a second predicate $p_{intransitive}$ is introduced and analyzed.

3.3.4. Intransitive Information-Flow Policies

The intuition behind intransitive information-flow policies is to authorize communication from a thread τ_H to a lower or incomparably classified thread τ_L only if this communication is relayed over a third thread τ_M. Thereby, the role of this third thread is to properly sanitize and filter the information it forwards. In intransitive information-flow policies, this fact is expressed by assigning τ_M a secrecy level $dom(\tau_M)$ for which $dom(\tau_H) \leq dom(\tau_M)$ and $dom(\tau_M) \leq dom(\tau_L)$ holds and by requiring $dom(\tau_H) \nleq dom(\tau_L)$. That is, $dom(\tau_M)$ is the intransitive point of the pass $(dom(\tau_H), dom(\tau_M), dom(\tau_L))$.

A cryptographic gateway [HWF05] (Figure 3.8) is an example scenario for intransitive information-flow policies. To securely connect two potentially untrustworthy subsystems over the Internet, each site runs an instance of the cryptographic gateway. This gateway is comprised of two components, a trusted wrapper and a legacy OS instance, which contains a not necessarily trustworthy network protocol stack. The purpose of the wrapper is to encrypt messages from the local subsystem before it forwards the encrypted messages to the local protocol stack. The protocol stack in turn transmits the encrypted messages to the remote site. In the reverse direction, the gateway decrypts the messages it receives from the local protocol stack. The gateway is trusted to properly sanitize and filter outgoing messages.

In the previous section, we have seen that $p_{transitive}$ suffices for transitive information-flow policies. However, the above argument, which supports this predicate, no longer holds for intransitive information-flow policies. More precisely, in the case $\neg p_{transitive}(\tau_h)$, we assumed that $dom(\tau_h) \leq dom(\tau_s)$ and $dom(\tau_s) \leq dom(\tau_r)$ implies $dom(\tau_h) \leq dom(\tau_r)$. This is no longer the case if $dom(\tau_s)$ is the intransitive point of the intransitive pass $(dom(\tau_h), dom(\tau_s), dom(\tau_r))$. In this case, $dom(\tau_h) \not\leq dom(\tau_r)$ holds.

The following example shows that, for intransitive information-flow policies, $p_{transitive}$ cannot prevent all covert channels due to direct and indirect influences. Let three threads τ_H, τ_M and τ_L be classified as described above (i.e., $(dom(\tau_H), dom(\tau_M), dom(\tau_L))$ is an intransitive pass). If $prio(\tau_L) > prio(\tau_H)$ and $prio(\tau_H) > prio(\tau_M)$, $\neg p_{transitive}(\tau_H)$ holds unless a further thread causes $p_{transitive}(\tau_H)$ to hold. Because $\{\tau_H, \tau_M\} \subseteq T_{low}(\tau_H)$ and $dom(\tau_H) \leq dom(\tau_H) \wedge dom(\tau_H) \leq dom(\tau_M)$, the scheduler runs τ_H unconstrained. But then τ_H is able to influence when τ_M's messages arrive at τ_L.

3.3.4.1. A Countermeasure Predicate for Intransitive Information-Flow Policies

One possible approach to address intransitive information-flow policies is to apply *Countermeasure I* more often. By removing the priority constraint in $p_{transitive}(\tau)$, we obtain the predicate:

Definition 12. Predicate for Intransitive Policies.
For intransitive information-flow policies, the following predicate determines whether Countermeasure I *(see Definition 10) has to be applied for the thread τ at the point in time t.*

$$p_{intransitive}(\tau, t) := \exists \tau' \in T.\ dom(\tau) \not\leq dom(\tau')$$

For intransitive information-flow policies, *Countermeasure I* with $p_{influence} = p_{intransitive}$ avoids information leakage due to direct and indirect influences. A corresponding machine-checked non-interference proof is described in Völp et al. [VHH08b]. The proof is for a simplified version of the proposed budget-enforcing fixed-priority scheduler. The PVS sources of this proof are published [VHH08a].

However, although $p_{intransitive}$ leads to a secure scheduler, the consequences of applying *Countermeasure I* based on this predicate are severe. $p_{intransitive}$ evaluates to false only for those threads τ that are classified at the lowest secrecy level $\perp$ (i.e., for which $dom(\tau) = \perp$ holds). All other threads are constrained by the countermeasure. That is, whenever a thread that has access to some secret blocks or stops, the scheduler will switch to the idle thread to consume this thread's total budget. Therefore, for threads that have access to some confidential information, the system is as restrictive as a time-partitioning system. The higher scheduling overhead of a fixed-priority scheduler is not justified.

3.3.4.2. Avoiding Intransitive Points

An alternative to applying *Countermeasure I* more often is the following restructuring of servers at intransitive points. For scheduling these restructured servers in a non-interference secure manner, a second information-flow policy can be extracted. Because this extracted policy is transitive, *Countermeasure I* $p_{influence} = p_{transitive}$ suffices to avoid leakage due to direct and indirect influence.

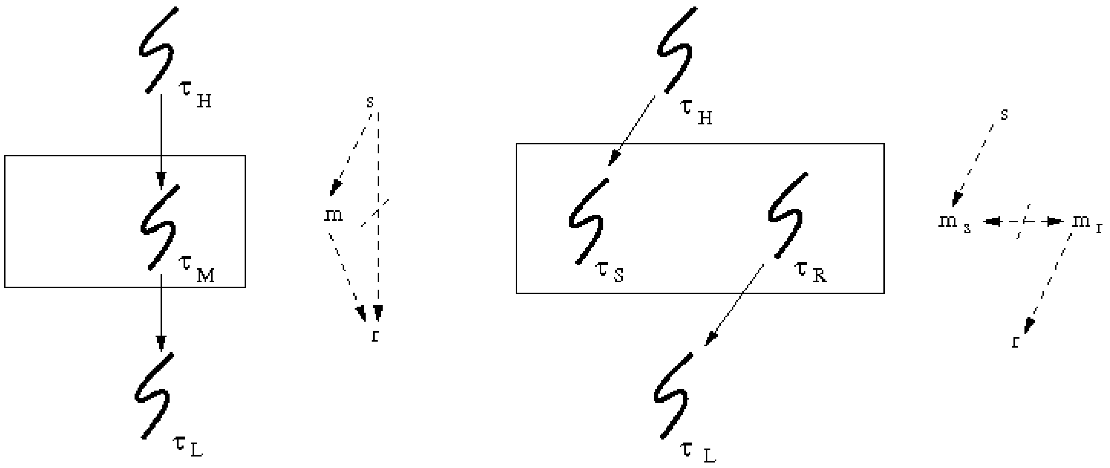

Figure 3.9.: The original, intransitive information-flow policy classifies the thread τ_M of the cryptographic gateway at secrecy level m. The clients of this gateway receive the secrecy levels s and r. Together, (s, m, r) is an intransitive pass (left picture). After restructuring the gateway, it contains two threads τ_S and τ_R. The extracted transitive information-flow policy classifies these threads at incomparable secrecy levels m_s and m_r (right picture).

To forward a sanitized message without leaking secrets that a sender encodes in the timing of a server, the sender must not directly influence the forwarding thread or cause another thread to do so. Let τ_H be the sending client thread, τ_L the receiver of the sanitized message and τ_R the server thread that forwards τ_H's message to τ_L. Then, either $dom(\tau_H) \not\leq dom(\tau_R)$ must hold or τ_R must run at a sufficiently high priority (see Section 3.2.1) to not be influenced. Because the latter negatively affects the response times of real-time threads, let us in the following focus on the first approach.

Clearly, $dom(\tau_H) \not\leq dom(\tau_R)$ prevents the server from being single threaded because τ_R must not receive the message from τ_H. Let us therefore assume that the server provides at least one thread for each differently classified client (e.g., τ_S to receive τ_H's messages and τ_R to forward them to τ_L). Then, if the server guarantees to suppress illegal information flows, the extracted information-flow policy can classify these threads such that $dom(\tau_H) \leq dom(\tau_S)$ and $dom(\tau_R) \leq dom(\tau_L)$ holds and that otherwise no information flows are allowed between these threads. This implies in particular that $dom(\tau_S) \not\leq dom(\tau_R)$.

A prerequisite for the guarantee to suppress illegal information flows is that server-internal synchronization and communication primitives are safe in the sense that a server thread cannot affect the externally observable timing behavior of another server threads if it invokes such a primitive. We shall return to such safe synchronization primitives in Section 3.7 on page 107.

More generally, the extraction of the second information-flow policy $(L', \leq', dom')$ works as follows. For each intransitive pass (s, m, r) in the original information-flow policy $(L, \leq, dom)$, two new, incomparable secrecy levels m_s and m_r are added disjointly to the set of secrecy levels L (i.e., $L' := L \uplus \{m_s, m_r\}$ where $m_s \not\leq' m_r \wedge m_r \not\leq' m_s$). Then, the pairs $s \leq m$ and $m \leq r$ are replaced by $s \leq' m_s$ and $m_r \leq' r$ in the dominates relation $\leq'$ of the restructured information-flow policy. Moreover, if s (respectively r) is not an intransitive point, any pair $s' \leq s$ is extended to $s' \leq' m_s$ and any pair $r \leq r'$ is extended to $m_r \leq' r'$. Because intransitive points may also transitively be connected such as in $x \leq m \wedge m \leq r \wedge x \leq r$, we also have to swing all remaining connections to the newly introduced secrecy levels. In the example, this means $x \leq' m_s$ and $x \leq' m_r$ is added to $\leq'$. Finally, the threads of the restructured server are classified to their new secrecy levels: $dom'(\tau_S) = m_s$ and $dom'(\tau_R) = m_r$. Figure 3.9 illustrates this transformation.

There are five important points to notice:

1. Because the set of secrecy levels L is finite, the above extraction algorithm terminates;
2. All previously authorized information-flows remain authorized except those between the newly introduced threads in the restructured servers;
3. A consequent application of this transformation for all intransitive points results in a transitive information-flow policy;
4. Communication from τ_S to τ_R is authorized outside this extracted information-flow policy; and hence,
5. Server-internal communication and synchronization must be safe despite influences from other threads.

For the cryptographic gateway, a simple strategy to avoid a leakage of timing information to the protocol stack is to produce an artificial message whenever the protocol stack expects such a message and no sender has provided one. Because messages are encrypted, the protocol stack cannot distinguish fake from real messages (provided of course, the encryption is sufficiently strong). The stack cannot deduce any information about the order and timing of the real messages.

The transitivity of the extracted information-flow policy follows from the following observation. If (s, m, r) is an intransitive pass in the original information-flow policy where neither s nor r are intransitive points, the transformation extends any relation $s' \leq s$ to $s' \leq m_s$ such that $s' \leq s \wedge s \leq m_s \Rightarrow s' \leq m_s$. Likewise, $r \leq r'$ is extended to $m_r \leq r'$ such that $r \leq r' \wedge m_r \leq r \Rightarrow m_r \leq r'$. If s is an intransitive point (e.g., of the pass (x, s, m)) then the transformation step of m replaces this pass with the intransitive pass (x, s, m_s) where m_s is no longer an intransitive point. Hence, a subsequent transformation of s will split the pass into $x \leq s_x$ and $s_m \leq m_s$, where s_x and s_m are no longer intransitive points. If r is an intransitive point, the transformation step of r proceeds accordingly. Because each transformation step removes one intransitive point and because the newly introduced secrecy levels are connected in a transitive way, the resulting information-flow policy is transitive.

Assuming that our envisaged open microkernel-based system has been transformed as described above, I will focus on transitive information-flow policies in the remainder of this chapter.

3.3.4.3. Implementing *Countermeasure I* with Static Predicates

The two static predicates $p_{transitive}$ and $p_{intransitive}$ allow for the following simple and well-performing implementation of *Countermeasure I*.

For a thread τ, both $p_{transitive}(\tau)$ and $p_{intransitive}(\tau)$ can be evaluated while the system is offline and without considering the point in time when *Countermeasure I* should be applied. Therefore, the result of evaluating one of these predicates can be stored in a flag in τ's thread control block.

Whenever the scheduler determines whether τ is ready, it can just read the countermeasure flag from τ's thread control block and treat this thread as ready if it is active and if its countermeasure flag is set. This implies that blocked or stopped threads or threads are not dequeued from the ready queue, which means the scheduler may select a blocked or stopped thread to run. To prevent this, the thread-switch procedure of the scheduler is modified to check whether the selected thread is actually ready or whether it is blocked or stopped and only treated-as-ready. In the latter case, the thread-switch procedure turns control to the idle thread after activating the original thread's total budget. As a consequence, the idle thread automatically consumes the total budget of this originally selected thread τ. The execution budget of τ is only activated if the thread-switch procedure turns control to τ.

The scheduling overhead for checking an additional flag in the thread control block is negligible. The thread control block must be accessed anyway to extract the state of the corresponding thread. Because the search for the next highest prioritized ready thread terminates if a treated-as-ready thread is found, the search takes no longer than on a system with an unmodified scheduler. A further benefit of the above implementation is that the kernel needs no knowledge about the information-flow policy.

3.3.5. Avoiding Information Leakage due to Non-preemptive Execution

To eliminate timing channels due to non-preemptive execution, the scheduler cannot just preempt possibly leaking threads. If threads execute non-preemptively to synchronize critical sections, forcibly preempting these threads would result in incorrect synchronization.

The following observation leads to a countermeasure to avoid timing-channels due to non-preemptive execution. In a real-time system, all non-preemptive code paths must have a bounded execution time. Otherwise, interrupt latencies are unbounded and no guarantees could be given for the response times of threads. Let max_delay_l be the kernel-enforced upper bound on the time that a thread τ_l can execute non-preemptively, which I have introduced in Section 3.1.2. Then, because a thread must be running to execute non-preemptively, only the currently running thread can defer the preemptions of higher prioritized threads. As long as the scheduler resumes the highest prioritized thread τ_h whose preemption is pending, only one non-preemptively executing lower prioritized thread can defer when τ_h resumes its execution: the lower prioritized thread that did run when τ_h's preemption occurred. Hence, the maximum time that a preemption of τ_h can be deferred is:

Definition 13. Maximum Preemption Delay
The maximum time a preemption of a thread τ_h can be delayed is

$$max\_delay\_low(\tau_h) := \max_{\tau_l \in T_{low}(\tau_h)} (max\_delay_l)$$

By delaying all preemptions of possibly leaked to higher prioritized threads by $max\_delay\_low(\tau_h)$, we obtain a second countermeasure to avoid leakage due to non-preemptive execution:

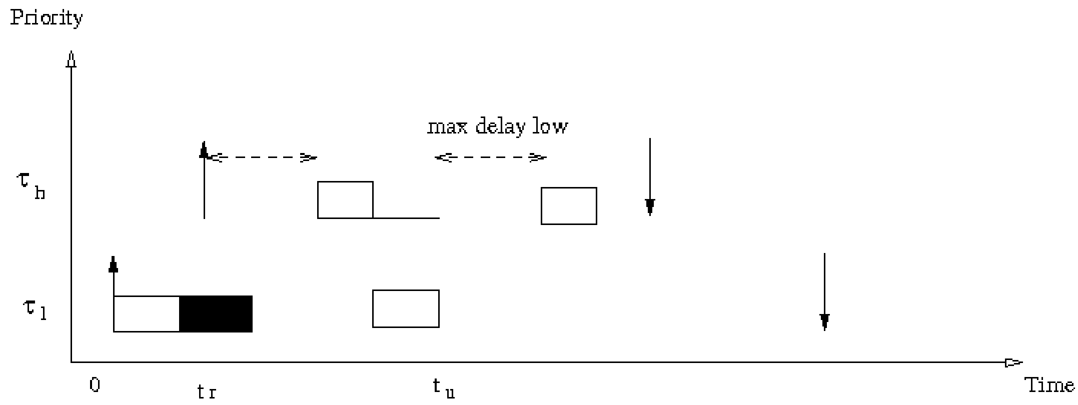

Figure 3.10.: The scheduler delays the two points in time when τ_h resumes its execution after the two preempting events at t_r and t_u by $max\_delay\_low(\tau_h)$. The thread τ_h can therefore no longer distinguish this countermeasure from the non-preemptive execution of τ_l.

Definition 14. Countermeasure II

Let $p_{delay}(\tau_h, t)$ be a predicate such that if $\neg p_{delay}(\tau_h, t)$ holds a thread τ_h resumes its execution immediately if a preempting event releases or unblocks τ_h at time t. If $p_{delay}(\tau_h, t)$ holds at this time, the following countermeasure is applied. The second countermeasure to avoid leakage due to non-preemptive execution is to delay any preemption of a thread τ_h, which is triggered by a preempting event at time t, by $max\_delay\_low(\tau_h)$ if $p_{delay}(\tau_h, t)$ holds. As a result of delaying this preemption, τ_h cannot resume its execution before $t + max\_delay\_low(\tau_h)$.

Figure 3.10 illustrates *Countermeasure II*. The release or the unblocking of a higher prioritized thread τ_h preempts the lower prioritized thread τ_l at t_r respectively at t_u. An unmodified scheduler would run τ_h immediately after τ_l stops executing non-preemptively (end of black bar) respectively immediately at t_u. To avoid information leakage due to non-preemptive execution if τ_l is not cleared to send to τ_h, *Countermeasure II* delays both preemptions. As a result, τ_h can resume its execution only after $t_r + max\_delay\_low(\tau_h)$ respectively after $t_u + max\_delay\_low(\tau_h)$.

The static predicate p_{delay} overestimates the points in time when the scheduler has to apply this second countermeasure. It is defined as follows:

Definition 15. Predicate for Countermeasure II.

A preemption at time t, which is caused by a thread τ_h, must be delayed to avoid information leakage due to non-preemptive execution if the following predicate holds:

$$p_{delay}(\tau_h, t) := \exists\, \tau_l \in T_{low}(\tau_h). \quad dom(\tau_l) \not\leq dom(\tau_h) \;\wedge\; max\_delay_l > 0$$

Like $p_{transitive}$ and $p_{intransitive}$, p_{delay} is also time-independent. I emphasize this independence because I will simply write $p_{delay}(\tau)$ to mean $p_{delay}(\tau, t)$. The predicate $p_{delay}(\tau_h)$ holds for transitive information-flow policies (see Section 3.3.4.2) whenever there exists a lower or equally prioritized thread τ_l that is not cleared to send to τ_h but that is able to execute non-preemptively (i.e., for which $max\_delay_l > 0$ holds).

The following observation gives an intuition why *Countermeasure II* with p_{delay} avoids information leakage due to non-preemptive execution. There are three types of leakage due to non-preemptive execution:

1. *direct leakage* from a non-preemptively executing thread τ_l to a higher prioritized thread τ_h,
2. *indirect leakage* from τ_l to another thread τ_r by influencing the timing of τ_s's messages to τ_r, and
3. leakage from τ_l to lower or equally prioritized threads that exploit two corner case situations.

Let me defer the discussion of these corner case situations to the next section.

Case 1: direct leakage

If $p_{delay}(\tau_h)$ holds, the scheduler delays all preemptions by $max\_delay\_low(\tau_h)$ that would release or unblock the thread τ_h. Therefore, and because a lower or equally prioritized thread τ_l can at most execute non-preemptively for $max\_delay_l \leq max\_delay\_low(\tau_h)$, τ_h cannot distinguish between a delay caused by *Countermeasure II* or a delay caused by τ_l executing non-preemptively. No information can be leaked directly from τ_l to τ_h.

Case 2: indirect leakage

If $p_{delay}(\tau_h)$ holds, τ_h's execution and hence the timing of its messages cannot be influenced by τ_l.

If $p_{delay}(\tau_h)$ does not hold, we can conclude from the transitivity of $\leq$ that for a receiver τ_r of τ_h's messages it holds that:

$$dom(\tau_l) \leq dom(\tau_h) \wedge dom(\tau_h) \leq dom(\tau_r) \Rightarrow dom(\tau_l) \leq dom(\tau_r)$$

The indirect information flow between τ_l and τ_r is not in violation to the system's information-flow policy. $dom(\tau_d) \leq dom(\tau_x)$ already authorizes τ_l to send to τ_r.

3.3.5.1. Corner Cases

Leakage due to non-preemptive execution is typically directed from a low-priority thread to a higher prioritized thread. However, there are two corner-case situations in which a non-preemptively executing thread τ_i can leak information to lower prioritized threads:

- if τ_i executes non-preemptively while exceeding the execution or total budget of its current job; or,
- if τ_i executes non-preemptively when the deadline of its current job has passed.

There are two principle approaches to prevent leakage at these end-of-release preemptions:

1. The scheduler can trigger the respective preemptions ahead of time; or,
2. It can forcibly preempt a thread at these points in time.

In the latter case, the scheduler has to export enough information to allow well-behaving threads to check whether it is safe to enter a critical section.

Because a thread τ_i can execute non-preemptively for at most max_delay_i, a scheduler can avoid both corner cases if it triggers end-of-release preemptions already when one of the remaining budgets falls below max_delay_i respectively when the adjusted relative deadline $d_{i,k} - max\_delay_i$ passes.

A scheduler that informs a thread τ_i about outstanding end-of-release preemptions ahead of time can forcibly preempt τ_i when one of these preemptions occur without risking the correctness of well-behaving threads. Because τ_i knows about these preemptions in advance, it can avoid entering a critical section in situations where insufficient time remains to complete this critical section.

To inform the currently running thread about imminent end-of-release preemptions, the scheduler can either setup a timer and signal the currently running thread when this timer expires or it can provide the thread with sufficient information to predict these preemptions itself.

To signal end-of-release preemptions ahead of time, the scheduler must setup two timers for all jobs $\tau_{i,k}$ that do not stop ahead of time: the first max_delay_i ahead of end-of-release preemptions, the second to actually deactivate this job. However, the overhead of the involved kernel entry would by far outweigh the benefit of a user-level delayed-preemption mechanism to synchronize short critical sections.

Avoiding Critical Sections by Predicting End-Of-Release Preemptions To predict when an end-of-release preemption occurs, the currently running job $\tau_{i,k}$ has to know its release point $r_{i,k}$, the time t_e when the scheduler has last switched to this thread and the minimum of its remaining execution and total budget at this time (i.e., the minimum of $eb_{rem,i}(t_e)$ and $tb_{rem,i}(t_e)$). The following pseudo code verifies that the remaining time before an end-of-release preemption is at least $|cs|$, where $|cs|$ is the maximal time required to execute the critical section cs.

```
retry:
  disable_preemptions();

  t = read_clock();

  if (eb_rem,i(t_e) + t − t_e < |cs| or
      tb_rem,i(t_e) + t − t_e < |cs| or
      r_i,k + d_i,k − t < |cs| or
      preemption_occurred)

      enable_preemptions();
      goto retry;

  // enter critical section
```

To avoid a preemption after a successful check but before the thread enters the critical section, preemptions must already be disabled during the check. Because the scheduler forcibly preempts threads when they exhaust their budgets or when the deadlines of these threads pass, the code must test for pending preemptions. If a preemption is pending, τ_i may have read an old version of $eb_{rem,i}(t_e)$ or $tb_{rem,i}(t_e)$ and retries the operation to avoid inconsistencies.

The costs for executing the above code snippet are dominated by the time required to read the system clock. The time required to access the exported values in the thread's UTCB and the time required for the conditional jump are negligible because the part of the UTCB that stores these parameters will likely remain in the first level cache of the processor and the jump can statically be predicted as not taken.

The total time required to execute the above code snippet is 9 cycles on an AMD Dual-Core (2.4 GHz) and 51 cycles on a Pentium M (1.6 GHz) [6]. For short critical sections such as an enqueue operation to the head of a double-linked list, these costs can be significant. On the AMD Dual-Core this list enqueue takes 4 cycles without and 13 cycles with the above check. On the Pentium M these are 6 cycles respectively 57 cycles.

As long as short critical sections are rare, the overhead of the check is tolerable although it increases the time of a list enqueue operation by an order of magnitude. Otherwise, schedulers that trigger preemptions ahead of time rather than signalling them are to be preferred. In real-time systems max_delay_i is typically in the order of a few 1000 cycles on modern GHz processors. In comparison to that, total budgets and relative deadlines are typically set to several hundred milliseconds. Hence, budget depletion and passing deadlines are rare events and the idle time due to the earlier preemption of a thread τ_i is negligible.

3.3.5.2. Implementation

In fully interruptible kernels, pre-located trampoline code allows for a straightforward implementation of *Countermeasure II*. Recall from Section 3.2.2 on page 59, a kernel is fully interruptible if it allows interrupts to preempt kernel code at any point in time. In such a kernel, the interrupt handler is invoked immediately when an interrupt occurs. It can therefore record the time $t_{preemption}$ of this occurrence and return to the interrupted code if this code has expressed its intent to execute non-preemptively. With $t_{preemption}$, *Countermeasure II* can be implemented by modifying the scheduler to switch to pre-located trampoline code instead of switching directly to thread τ_h that has caused this preemption. If the trampoline code gets activated (e.g., after the currently running thread has stopped executing non-preemptively) for a thread τ_h with $p_{delay}(\tau_h)$, it computes the earliest absolute point in time when τ_h can resume its execution as $t_{preemption} + max\_delay\_low(\tau_h)$. After that it idles until this point in time has passed at which time it returns control to τ_h.

Although the point in time when a processor stops executing the currently running thread is available at the architectural level, it is currently not exported to the operating-system kernel. This point in time is precisely the time when the last instruction of the current thread retires before the processor activates the in-kernel interrupt handler.

As long as the times for entering the kernel and for activating an in-kernel interrupt handler are sufficiently constant, the interrupt handlers of fully interruptible kernels can approximate $t_{preemption}$ by reading the system clock immediately when they start executing. On processors where these times vary significantly, low-bandwidth covert channels can remain.

This completes the discussion of the budget-enforcing fixed-priority scheduler and of its variants for transitive and intransitive information-flow policies. So far, we have assumed distinct thread priorities and the idle thread to consume the total budgets of blocked or stopped treated-as-ready threads. In the following, I shall lift these restrictions first by considering other budget-consumer threads and then, in Section 3.3.8, by extending the two countermeasures to equally prioritized threads.

[6] Appendix A contains the C++ source code for this check and for the measured list-enqueue operation.

3.3.6. Accounting

For *Countermeasure I* to work (see Definition 10 on page 61), the scheduler must accurately account both *Countermeasure I* and *Countermeasure II* to the two budgets of a thread. Otherwise, variations in the imprecise remaining total budget could be exploited to leak confidential information. Obviously, *Countermeasure I* must be accounted to the total budget of the treated-as-ready thread. *Countermeasure I* treats possibly leaking threads as if they were ready and runs the idle thread as a budget consumer when the treated-as-ready thread blocks or when it has stopped.

Less obvious is where to account the time of the second countermeasure. Intuitively, one would account the time that a thread τ_l executes non-preemptively to this thread's execution budget. However, preemptions of a higher prioritized thread τ_h are also deferred in situations where no lower prioritized thread executes non-preemptively or where no such thread runs at all.

A partial accounting to both τ_l and τ_h (e.g., first to τ_l's execution budget as long as it executes non-preemptively and then to τ_h's total budget) must be rejected as well because such an accounting reveals information about the non-preemptive execution of τ_l in the remaining budget of τ_h. Partial accounting gives rise to a covert channel.

To avoid these channels, we have to account *Countermeasure II* entirely to τ_h's budgets. More precisely, as long as τ_h is the highest-prioritized ready or treated-as-ready thread, it consumes its total budget even if a lower prioritized thread executes non-preemptively and even if *Countermeasure II* delays its preemptions. Whether the scheduler accounts the countermeasure only to τ_h's total budget or whether it accounts this countermeasure to both τ_h's execution and total budget is irrelevant as far as information-flow security is concerned. An accounting to τ_h's execution budget is however the more natural choice because no lower prioritized thread (except the current non-preemptively executing thread) can run while the scheduler applies this countermeasure.

Notice, to correctly account *Countermeasure II* to τ_h's execution budget, the kernel must interrupt the non-preemptively executing threads to record the point in time when a preempting event has occurred. Preempting events are therefore visible to non-preemptively executing threads τ_l. Still, the scheduler is non-interference secure:

1. The unblocking of a higher prioritized thread τ_h (though not its release) preempts a lower prioritized thread τ_l only if the scheduler has selected τ_l. This can only happen if $\neg p_{transitive}(\tau_h)$. But then $dom(\tau_h) \leq dom(\tau_l)$ and τ_l is authorized to see the preempting events of τ_h;

2. The release of a higher prioritized thread τ_h can preempt a lower prioritized thread τ_l. However, lower prioritized threads are assumed to be cleared to these releases.

3.3.7. A Budget-Enforcing Fixed-Priority Lattice Scheduler

So far, the idle thread played the role of the budget-consumer thread to avoid leakage due to direct and indirect influences. However, the idle thread performs no useful work. In the following, I will therefore explore the possibility to use other threads as budget-consumer threads.

Clearly, such a thread τ_H (i.e., a thread for which $dom(\tau_H) = H$ holds) must be ready and its secrecy level must dominate the secrecy level of the selected thread τ_L (i.e., $dom(\tau_L) \leq dom(\tau_H)$). Otherwise, τ_L could leak confidential information to τ_H by modulating when it blocks and thereby when the scheduler selects τ_H to consume τ_L's total budget. If τ_L must not send to a lower prioritized thread τ_r, then τ_H must also not send to τ_r. Otherwise, if $dom(\tau_H) \leq dom(\tau_r)$, we could immediately conclude from the transitivity of the information-flow policy that $dom(\tau_L) \leq dom(\tau_r)$, which authorizes τ_L to send to τ_r in the first place.

If τ_H blocks while consuming τ_L's total budget, the problem of finding a suitable budget-consumer thread recurs. In this case, the scheduler must find another ready thread τ'_H to consume τ_L's budget. To avoid leakage both $dom(\tau_L) \leq dom(\tau'_H)$ and $dom(\tau_H) \leq dom(\tau'_H)$ must hold. Otherwise, either τ_L or τ_H could leak to τ'_H by modulating when they block. Fortunately, for transitive information flow policies, the latter condition $dom(\tau_H) \leq dom(\tau'_H)$ implies the former. Therefore, if a budget consumer blocks, the scheduler just needs to search for another ready thread that is higher classified than the last budget consumer. Because $dom(\tau_{idle}) = \top$ and because the idle thread never blocks, the scheduler will always find a budget consumer for τ_L. Following Hu [Hu92], I call a scheduler that implements this search strategy a *budget-enforcing fixed-priority lattice scheduler*.

Notice that all threads τ^i_H consume τ_L's total budget. Notice further that budget consumers have a lower priority than τ_L. Otherwise, they would have preempted τ_L and the scheduler would have selected these threads in the first place. Hence, $p_{delay}(\tau_L)$ already considers these threads. Budget consumers cannot leak information to τ_L by executing non-preemptively.

Naturally, running a budget consumer τ^i_H can change how this thread behaves in the future. However, to observe this change a thread τ_X must be cleared to see τ^i_H. Because $\leq$ is transitive, τ_X is then also cleared to all previous budget consumers $\tau^j_H, j < i$ and to τ_L.

3.3.7.1. Implementation

The search for a budget-consumer thread can be implemented in various different ways:

- the scheduler can search the ready list for a lower prioritized thread with higher or equal secrecy level;
- it can maintain an additional ready list, which is sorted by ascending secrecy levels; or,
- it can maintain for each thread a pointer to a potential budget consumer. If this thread is ready, it becomes the new budget-consumer thread. If not, the search proceeds by chasing these pointers until a potential budget consumer is found that is ready.

For microkernels, the last alternative is most attractive because the kernel needs no knowledge about the information-flow policy. In particular, when this policy changes frequently, enforcement mechanisms that require no knowledge about this policy are preferable.

3.3.8. Limited Number of Priorities

In real-life systems, the number of distinct priority levels are limited (typically to between 16 and 256 levels). So far, I have assumed that threads have distinct priorities (see Section 3.3 on page 60). However, if more threads run in a system than there are priority levels, multiple threads have to share the same priority level. In this case, a second scheduling policy selects the thread that should run next if no higher prioritized thread is ready (or treated-as-ready). The two most prominent scheduling policies for threads sharing the same priority level are *first-come-first-served* (FIFO) and Round Robin.

In real-time systems literature [CDKM02, But05], FIFO is often defined only for threads that do not block. In this case, threads of the same priority complete in the order in which they are released. The POSIX System API Realtime Extension [IEE93] is an exception. It defines FIFO in terms of a list of ready threads [7]. Threads are enqueued to this list in the order of their release. However, blocked threads are removed from this list and re-enqueued at its tail when they unblock. Although a precise definition of FIFO with blocked threads is missing in the cited version of her book, Liu [Liu00] seems to assume that blocked threads keep their list position [8]. To distinguish Liu's version of first-in-first-out from the version described in the POSIX Real-Time Extensions, I will call the former version FIFO and the latter version POSIX-FIFO.

Round Robin works very similar to FIFO except that execution budgets of threads are split into smaller chunks. The Round Robin scheduler then schedules the threads according to one of the FIFO versions until their current chunk depletes. In this case, the scheduler refills the chunk and re-enqueues the thread at the end of the FIFO queue. This refill happens until the execution budget of a thread is depleted respectively until the deadline of this thread passes. Before threads are re-scheduled by a Round Robin scheduler, they have to wait for all other threads to complete one chunk.

If multiple threads share a single priority level, direct and indirect influence as well as influences due to non-preemptive execution can affect also equally prioritized threads. Which threads are

[7] Implementations are of course free to choose a different data structure.

[8] I draw this conclusion from the time-demand function Liu presents in Section 6.8.4: "Priority-Driven Scheduling of Periodic Tasks – Practical Factors – Limited-Priority Levels".

In Section 3.3.1, Liu introduces the execution time e_k of a task (in our terminology a thread) as the maximum execution time of all its jobs. For the time demand $w_i(t)$ of a thread τ_i, Liu gives the following formula: $w_i(t) = e_i + b_i + \sum_{T_k \in \mathbf{T}_E(i)} e_k + \sum_{T_k \in \mathbf{T}_H(i)} \lceil \frac{t}{p_k} \rceil e_k$. It says that in addition to the usual time required to schedule the thread τ_i (first two terms) and simultaneously released higher prioritized threads (last term), a time demand of $\sum_{T_k \in \mathbf{T}_E(i)} e_k$ is required. This additional third term seeks to characterize the worst-case time demand of threads executing at the same priority.

Let us assume a system with two equally prioritized simultaneously released threads τ_1 and τ_2. Assume further that τ_2's period Π_2 is twice as large as τ_1's period (i.e., $\Pi_2 = 2\Pi_1$). Assume also that τ_1 never blocks and that both jobs of τ_1 execute for e_1 milliseconds.

If τ_2 keeps its list position while blocked, only the first job of τ_1 can execute before τ_2. I assume here that the scheduler inserts $\tau_{1,1}$ before $\tau_{2,1}$ into the FIFO list. The second job of τ_1 is released after τ_2 and will therefore reside after τ_2 in the FIFO list. It can execute only during those times when τ_2 blocks. These are however, bounded by the second addend: the worst-case blocking time b_2.

More generally, at most one job of each equally prioritized thread can execute before a thread τ_i. The term $\sum_{T_k \in \mathbf{T}_E(i)} e_k$ correctly captures this fact.

On the other hand, if τ_2 has to re-enter the FIFO list at the tail, both jobs of τ_1 can execute before τ_2 completes unless τ_2 wants to execute for less than $\Pi_1 - e_1$ milliseconds. In this situation, the additional time demand of τ_2 is $2e_1$. That is, twice the amount that $w_i(t)$ considers.

To conclude, Liu assumes either

1. that equally prioritized threads also have equal periods (an assumption Liu abandons immediately after Section 6.8.4), or
2. she assumes that threads never block during their execution (which is unlikely given the definition of b_i), or
3. she assumes that blocked threads keep their list position.

Of these assumptions, the last is most likely as it is consistent with the other parts of Liu's book.

effected by these influences depends on the scheduling policy for equally prioritized threads. In the following two sections, I discuss direct and indirect influences on equally prioritized threads and influences due to non-preemptive execution for FIFO, POSIX FIFO and two corresponding versions of Round Robin.

3.3.8.1. Direct and Indirect Influence of Equally Prioritized Threads

A thread τ_i that is scheduled according to FIFO (Liu's version) can directly influence later released threads. The blocking behavior of τ_i cannot affect earlier released threads because these threads maintain their list position and are always selected before τ_i is considered. Unless a later released thread executes non-preemptively (see below) it is preempted once an earlier released thread resumes its execution.

POSIX FIFO re-enqueues blocked threads at the tail of the FIFO list. Therefore, a later released thread τ_j can also affect an earlier released thread τ_i when this thread has finished its blocking.

Unless a thread τ_i runs and blocks for at most the duration of one chunk, both versions of Round Robin allow a thread τ_j to directly influence τ_i even if τ_i never blocks. In this case, the execution and blocking behavior of τ_j influences how early τ_i can run its next chunk.

Countermeasure I avoids also direct and indirect influences of equally prioritized threads. Because a possibly leaking active thread τ_j is treated-as-ready, it maintains its position in the FIFO list until its deadline passes, until its total budget depletes or, in the case of Round-Robin, until the current chunk of its total budget depletes. Consequently, later-enqueued threads cannot distinguish whether τ_j executes or whether another thread consumes τ_j's budget.

The set $T_{low}(\tau_h)$ in the definition of the countermeasure predicate $p_{transitive}(\tau_h)$ (Definition 11) already contains equally prioritized threads. However, for FIFO, a more optimistic predicate than $p_{transitive}$ is imaginable. A thread τ_j can only directly influence later released threads. Therefore, if the order in which threads are released is known at admission time, only later released threads need to be considered. In general however, the release points and hence the order in which threads are released are not known at admission time.

3.3.8.2. Influence due to Non-Preemptive Execution of Equally Prioritized Threads

In FIFO (Liu's Version), blocked threads maintain their list position. As a consequence, earlier released threads can preempt a currently running later released thread of the same priority unless this thread executes non-preemptively. Conversely, later-released threads can delay the preemptions of earlier released threads by executing non-preemptively, which constitutes a covert channel.

Surprisingly, these channels cannot occur in POSIX FIFO and in the corresponding version of Round Robin. Provided that non-preemptively executing threads cannot defer end-of-release preemptions, and provided a similar precaution prevents threads from deferring end-of-chunk preemptions, later-enqueued threads cannot influence earlier-enqueued threads because if an earlier-enqueued thread blocks it is re-enqueued after the later-enqueued thread. Therefore, it cannot preempt this thread, which means leakage due to non-preemptive execution cannot occur between threads of this priority.

For POSIX FIFO and for POSIX Round Robin, the countermeasure predicate for *Countermeasure II* needs to consider only strictly lower prioritized threads. For FIFO, all equally prioritized threads must be considered unless the order of thread release points is known at admission time. The predicate $p_{delay}(\tau)$ (see Definition 15) checks for the existence of a possibly leaking thread in the set of all lower or equally prioritized threads $T_{low}(\tau)$.

3.3.9. Internal-Timing Channels

In our envisaged open microkernel-based system the scheduler must only eliminate external-timing channels in order to avoid all software-centric covert timing channels. This is because software-centric internal timing channels cannot exist in the unchecked and thus potentially untrustworthy single-level programs and because we assume a timing-leak transformation to eliminate internal timing leaks from the checked multi-level servers.

A single-level program cannot contain internal timing leaks although it may well encode information in the timing of externally observable events. However, because single-level programs can access only lower or equally classified information, these information flows cannot violate the information-flow policy. Legitimate observers of the events of a single-level program are already cleared to the secrecy level of the program and hence to all secrecy levels the program may read from.

This concludes the discussion of non-interference secure fixed-priority schedulers. Before I introduce the machine-checked non-interference proof for the budget-enforcing fixed-priority scheduler, let us briefly consider proportional-share schedulers.

3.3.10. Information-Flow Secure Proportional-Share Schedulers

Although most L4-family microkernels implement fixed-priority schedulers, it is interesting to briefly discuss the information-flow properties of proportional share schedulers .

3.3.10.1. Basic Version of the Lottery and Stride Scheduler

The basic version of the stride scheduler and the basic version of the lottery scheduler are both non-interference secure.

For both schedulers, the decision to run a thread depends only on the tickets it holds. If the selected thread blocks, the system idles for one unit of time. Therefore, variations of a threads execution and blocking behavior have no influence on the point in time when other threads are scheduled.

Non-interference of the basic versions of the stride scheduler and of the lottery scheduler is preserved even if threads forward some of their tickets to legitimate receivers. Assume a thread τ_H (with $dom(\tau_H) = H$) forwards some of its tickets to a threads τ_X. If a thread τ_L is not authorized to receive information from τ_H it cannot distinguish whether τ_H or whether τ_X runs on one of τ_H's tickets. To detect that τ_X has received additional tickets, τ_L must be authorized to receive messages directly or indirectly from τ_X (i.e., $dom(\tau_X) \leq dom(\tau_L)$ must hold). However then, τ_L may legitimately receive information from τ_H because $dom(\tau_H) \leq dom(\tau_L)$ follows from the transitivity of $\leq$ and from the requirement that tickets can only be forwarded to legitimate receivers (i.e., $dom(\tau_H) \leq dom(\tau_X)$ must hold).

3.3.10.2. Compensate Tickets

Compensate tickets give rise to the following covert channel. While a thread τ blocks, other threads receive a larger share of the processor because the lottery scheduler and the stride scheduler repeat their selection procedure until they find a ready thread. When τ resumes its execution, it receives the compensation tickets. As a result, the share of other threads is temporarily reduced. Because threads may sample the shares they receive, they can detect variations in the execution and blocking behavior of other threads.

Countermeasure I (see Definition 14 on page 70), that is to treat possibly leaking threads as if they were ready, also works for proportional-share schedulers. If the scheduler selects the ticket of a possibly leaking thread τ, it runs a budget-consumer thread if the selected thread is blocked or if it has stopped. Therefore, when τ resumes its execution, it will not receive any compensate ticket. However, because compensation tickets affect the share of all threads, the predicate which activates this countermeasure must hold for all threads that are not classified at the lowest secrecy level $\perp$. In practice, a such constrained proportional-share scheduler achieves no benefit over the respective basic version without compensate tickets.

3.3.10.3. Proportional-Share Workloads on the Budget-Enforcing Fixed-Priority Scheduler

In Section 3.1.3.5, we have seen a possible mapping of proportional-share workloads to *ReThMo*. In this mapping, thread priority is a free parameter, which we can use to minimize the time when *Countermeasure I* has to be applied.

If the dominates relation $\leq$ of a transitive information-flow policy is a total order of secrecy levels (i.e., $\forall l, l'.l \leq l' \vee l' \leq l$ holds), *Countermeasure I* is never enabled if we assign thread priorities proportional to this order. That is, if $dom(\tau_h) \leq dom(\tau_l)$ holds for two threads τ_h and τ_l, they are assigned priorities that fulfil $prio(\tau_h) > prio(\tau_l)$.

If $\leq$ orders the set of secrecy levels only partially, we may have to apply *Countermeasure I* for some threads. If $\leq$ is a partial order, threads can exist for which neither $dom(\tau_m) \leq dom(\tau'_m)$ nor $dom(\tau'_m) \leq dom(\tau_m)$ holds. To prevent leakage due to direct and indirect influence, the scheduler must activate *Countermeasure I* at least for one of these threads: the one we choose to run at a higher priority. To minimize the time *Countermeasure I* is active we therefore have to select the thread with the smaller share to receive a higher priority.

Obviously, a setting of $tb_{i,k} = \Pi$ no longer works for threads τ_i for which $p_{transitive}(\tau_i)$ holds: *Countermeasure I* would only run τ_i and the budget-consumer thread but no thread that is lower prioritized than τ_i. Therefore, we have to limit the total budgets of τ_i to the share it receives. That is, $tb_{i,k} = prop_i\, \Pi$ for all jobs $\tau_{i,k}$ of τ_i.

3.4. A Machine-Checked Proof of Non-interference

In the following section, I describe the PVS-based non-interference proof for the proposed budget-enforcing fixed-priority scheduler in the variant described below. First, I briefly introduce the formalization of the model and the key concepts of this proof. In Sections 3.4.2ff, I will then revisit the individual parts of this model and describe the proof in greater detail.

The non-interference proof covers the following variant of the budget-enforcing fixed-priority scheduler:

1. the scheduler implements both *Countermeasure I* and *Countermeasure II*;
2. the information-flow policy is assumed to be transitive, that is, $p_{transitive}$, as introduced in Definition 11, is used to activate *Countermeasure I*, p_{delay}, as introduced in Definition 15, is used to activate *Countermeasure II*;
3. the idle thread serves as budget-consumer thread for *Countermeasure I*;
4. threads are not assumed to have distinct priorities. Equally prioritized threads are scheduled according to FIFO. That is, blocked threads maintain their position in the FIFO list;
5. non-preemptively executing threads cannot delay end-of-release preemptions;
6. *Countermeasure I* is accounted to the treated-as-ready thread's total budget, *Countermeasure II* is accounted to the highest-prioritized treated-as-ready thread as described in Section 3.3.6;
7. all threads are assumed to be cleared to the release points, deadlines and total budgets of higher or equally prioritized threads.

A corresponding proof for a simplified version of this scheduler and for $p_{influence} = p_{intransitive}$ is available in Völp et al. [VHH08a]. The PVS sources of the model and of the non-interference proof are published [Völ10]. Both are based on an abstract model of the scheduler. To extend these results to an actual implementation, the implementation must be shown to refine the abstract model in a confidentiality-preserving way [HPS01]. Extending the model to other versions of the scheduler should be straightforward.

3.4.1. Overview

The machine-checked proof of non-interference is based on a formal model of the above variant of the proposed budget-enforcing fixed-priority scheduler. I have formalized the behavior of this scheduler as a discrete-time state-transitioning system. Given an absolute point in time t and a state s as inputs, the state-transitioning system produces a new state s' that corresponds to the absolute point in time $t+1$. Without loss of generality, assume $t = 0$ for the initial state s_0. Although this state-transitioning system operates on discrete time values of type **Time : Type = nat**, it can produce schedules at any desired accuracy. This is because I do not specify how much time passes between two successive points in time t and $t + 1$, which means we can choose it to be arbitrary small.

State In the formal model, the state s keeps track of the scheduling parameters of the *ReThMo* task model (see Section 3.1.2 on page 48). The type of s is **State**. More precisely, **State** is a function type, which relates each thread to a record containing its non-constant scheduling parameters. In the non-interference proof, some of these parameters are left arbitrary but fixed. As a result, the proof holds for all possible settings of these parameters.
The formal model of the scheduler abstracts from certain implementation details such as the encoding of scheduling parameters in the thread control blocks, the various timers, which the scheduler programs to regain control, and the ready queue, in which the scheduler keeps all ready or treated-as-ready threads. The fundamental properties of these implementation details

are however preserved. For example, instead of formalizing timeouts, the model maintains a number of down-counting clocks, which indicate when such a timeout would fire. The ready list is replaced by a state-dependent predicate, which evaluates to true for the highest-prioritized ready or treated-as-ready thread of the given state, respectively for the first thread in the FIFO list if multiple threads share the same priority. For the proof of non-interference, only the total budget is relevant. The formal model of the scheduler therefore abstracts from the execution budget of a thread.

State Transformers State transformers encode the transitions that the scheduler representing state-transitioning system makes. A state transformer is a function of type:

```
State_Transformer : Type = [[State, Time] → State]
```

The input parameters of a state transformer are the state s and an absolute point in time t. To reduce the complexity of both, the model and the proof, I have formalized the scheduler as eight separate state transformers. Each of these state transformers stands for a specific scheduling decision. Together, they produce the result state s', which corresponds to the absolute point in time $t + 1$.

In a real-life system, scheduling decisions are triggered by timeouts, by explicit invocations of the scheduler in system calls (e.g., in blocking IPC) or by external interrupts that trigger the release or unblocking of a thread. In the formal model, I abstract from these specific invocations. Instead, the state transformers, which implement these scheduling decisions, are invoked for every absolute point in time t. When invoked, they check whether a triggering event has occurred and update the state s accordingly. For example, to produce the state for time $t + 1$, the state transformer that is responsible for releasing threads — **release_thread** — checks the release points of all threads to determine which threads are released at time t and modifies the state s accordingly.

The state transformer **dispatch_step** combines all eight state transformers. The function **dispatch** invokes **dispatch_step** recursively to produce the result state s' for the absolute point in time $t+1$ from a given initial state s_0. Quantification over t gives the desired universal result for all points in time.

Non-interference Non-interference formalizes the complete absence of security-policy violating information flows as the indistinguishability of observable outputs of a system. Provided that hardware covert channels have been addressed by other means, users can observe the microkernel-based system only through the threads that execute on their behalves. Given access to precise clocks, such a thread τ can report all those times when the scheduler runs τ or when it runs other threads that are authorized to send to τ. That is, an l-classified observer may learn about all points in time when a thread τ with $dom(\tau) \leq l$ executes, blocks or stops. The function **output(l,s)** extracts this information from the state s. It returns **nil** whenever no such thread runs and the state of τ if $dom(\tau) \leq l$ holds and τ is the highest-prioritized ready thread (respectively the first such thread in the FIFO list).

A scheduler is non-interference secure if for all l-similar initial states s_0 and s'_0 and for all points in time t it holds that:

$$output(l, dispatch(s_0, t)) = output(l, dispatch(s'_0, t)) \tag{3.4}$$

Informally, two initial states s_0 and s'_0 are l-similar if they agree on the parameters of those threads that an l-classified observer may see. Recall from Section 3.3 that threads τ are cleared to the release points, deadlines and total budgets of higher or equally prioritized threads. See also Point 5 in the above list at the beginning of this section. Therefore, l-similar states must also agree on these parameters if an l-classified observer is cleared to such a thread τ.

The intuition behind Equation 3.4 is the following. Assume an l-classified observer is able to see the points in time when l-observable threads τ execute preemptively or non-preemptively (denoted by **output**). If this observer cannot distinguish any two schedules, which origin from two l-similar initial states s_0 and s'_0 and which are produced by $dispatch$, then higher or incomparably classified threads cannot influence the execution of l-observable threads. Therefore, higher or incomparably classified threads cannot leak information over scheduling-related timing channels. The scheduler is non-interference secure with regards to this observer.

Proof The non-interference proof of the budget-enforcing fixed-priority scheduler proceeds in two stages: First, I show that the following property about pairs of states (s_t, s'_t) is invariant for the individual scheduler steps if they origin from two l-similar initial states s_0 and s'_0. The main non-interference result then follows from the observation that pairs of states, which fulfill this property produce the same outputs as seen by an l-classified observer.

Let $s_t = dispatch(s_0, t)$ and $s'_t = dispatch(s'_0, t)$ for the pair of l-similar initial states (s_0, s'_0) and for some point in time t. Then $s_{t+1} = dispatch\_step(s, t+1)$ and $s'_{t+1} = dispatch\_step(s', t+1)$ is a new pair of states for the point in time $t+1$. A property P about pairs of states is invariant if it holds for (s_0, s'_0) and if we can conclude from $P(s_t, s'_t)$ that $P(s_{t+1}, s'_{t+1})$ holds as well. The property of interest for the non-interference proof states:

1. that l-observable threads agree on their dynamic scheduling parameters; and

2. that if a low-priority thread τ is legitimately observable by an l-classified observer, then all higher and equally prioritized threads agree on the jobs they execute, on the remaining total budgets of these jobs, on the time that remains to the deadlines of these jobs and on the thread states of these jobs as far as activity is concerned. That is, either such a job is active in both states of the pair or it is inactive in both states.

The following four sections discuss the above ingredients of the scheduling model and of the non-interference proof in greater detail.

3.4.2. State

This section details the formalization of the state of the scheduler. I formalize the scheduling-related state of a thread with three record types:

- **Dynamic_State**,
- **Constant_Secrecy_Independent_State**, and
- **Constant_Secrecy_Dependent_State**.

The type **State** is an alias for **State : Type = [Thread → Dynamic_State]**. A corresponding mapping from **Thread** to the last two records are passed as additional input parameters to the eight state transformers.

The primary purpose of this split is to speed up the verification process by avoiding trivial results that the state transformers do not modify parameters in the last two records.

Later in this chapter, I will introduce a relation on pairs of scheduler states called l-similar. For now notice that two l-similar scheduler states agree on the parameters in the record type

Constant_Secrecy_Independent_State. The parameters in the record type **Dynamic_State** and in the record type **Constant_Secrecy_Dependent_State** may however differ for those threads that are classified at a higher secrecy level than l or are classified at a secrecy level that is incomparable to l.

In the following, I describe the above three record types in greater detail. For better readability I use the types **Job**, **Time** and **TimeSpan** as aliases for the type **nat**.

3.4.2.1. Dynamic State

```
Dynamic_State : Type = [#
  job                    : Job,
  remaining_total_budget : TimeSpan,
  remaining_deadline     : TimeSpan,
  remaining_max_delay    : TimeSpan,
  thread_state           : Thread_State,
  effective_release      : Time
#]
```

The parameter **job** denotes the job of a thread τ_i that is currently active respectively that is waiting for its release point if τ_i is currently inactive. The parameters **remaining_total_budget** and **remaining_deadline** are two down-counting clocks for $tb\_rem_{i,k}$ and for the time until the deadline $d_{i,k}$ passes. The parameter **remaining_max_delay** is a down-counting clock, which records the time that remains before the thread has executed non-preemptively for max_delay_i. In **thread_state**, I store whether the thread is **Ready**, **Blocked**, **Stopped** or **Inactive**. A thread is running if it is the highest prioritized ready thread and if it is at the head of the FIFO list. The latter condition is captured by **higher_effective_priority** (see Section 3.4.3.1 below). A thread executes non-preemptively (i.e., it is delaying) if it is running and if **remaining_max_delay** is positive.

The type **Thread_State** contains one state — **Delayed** — that is not one of the thread states of the *ReThMo* task model (see Section 3.1.2). The state transformers transition a thread into this state whenever *Countermeasure II* is active for this thread to avoid leakage due to non-preemptive execution (see Definition 14 on page 70). The scheduler defers the points in time when a **Delayed** thread resumes its execution after it is released and after it unblocks. The down-counting clock **effective_release** denotes how long a thread τ has to remains in this state. That is, **effective_release** denotes the time that remains until $t_{preemption} + max\_delay\_low(\tau)$ (see Section 3.3.5.2). The type **State** is defined as: **State : Type = [Thread → Dynamic_State]**.

3.4.2.2. Constant Secrecy-Independent State

```
Constant_Secrecy_Independent_State : Type = [#
  prio          : Priority ,
  label         : Label,
  release_label : {l : Label | l ≤ label},
  max_delay     : TimeSpan,
  max_delay_low : TimeSpan
#]
```

The parameters **prio**, **max_delay**, and **max_delay_low** store the respective parameters of the *ReThMo* task model (see Section 3.1.2). The parameter **label** is the secrecy level of the thread. The **release_label** of a thread is the secrecy level of its existence. Threads that are classified at a secrecy level, which dominates the **release_label** of a thread are cleared to observe the release points, deadlines and total budgets of this thread. The type of **release_label** is {**l : Label** | **l** ≤**label**}. This dependent type ensures that all threads are cleared to their own existence.

The eight state transformers take as an implicit parameter the PVS constant:

```
cts : (well_formed_constant_secrecy_independent_state)
```

The type of **cts** is a predicate subtype of **[Thread → Constant_Secrecy_Independent_State]**. The predicate for this subtype is:

```
well_formed_constant_secrecy_independent_state
                          (cts : [Thread → Constant_Secrecy_Independent_State]) : bool =
  (Forall (t_l , t_h : Thread) :
    cts(t_l)`prio ≤cts(t_h)`prio ⟹ cts(t_h)`release_label ≤cts(t_l)`label) ∧
  (Forall (t_l , t_h : Thread) :
    cts(t_l)`prio ≤cts(t_h)`prio ⟹ cts(t_l)`max_delay ≤cts(t_h)`max_delay_low)
```

The first clause of this predicate formalizes the assumption that lower prioritized threads are cleared to the release-points, deadlines and total budgets of higher prioritized threads (Point 5 of the list on page 79).

The second clause encodes the definition of $max\_delay\_low(\tau)$ (Definition 13 on page 69). It states that **max_delay_low** is an upper bound on the maximum time that lower prioritized threads can contiguously execute non-preemptively.

3.4.2.3. Constant Secrecy-Dependent State

```
Constant_Secrecy_Dependent_State : Type = [#
  release_point   : [Job  →  Time],
  deadline        : [Job  →  posnat],
  total_budget    : [Job  →  TimeSpan],
  action          : [Time →  Action]
#]
```

The record **Constant_Secrecy_Dependent_State** stores those constant thread parameters that may differ in l-similar scheduler states for higher than l or incomparably classified threads. These parameters are the release points $r_{i,k}$, the deadlines $d_{i,k}$ and the total budgets $tb_{i,k}$. Positive deadlines ensure that a thread is released for at least one unit of time before it gets deactivated by its deadline.

The parameter **action** encodes the action trace of a thread τ. The type of **action** is **[Time → Action]** where **Action** is a type which contains all the actions a thread can perform. These are: **block**, **stop**, **run**, and **run_non_preemptively**.

For the trace **action**, the type **[Time → Action]** is rather unconventional. However, because the state transformers use the parameter **action** only in a very restricted form, we can reap benefit of this simple formalization. There are three situations in which the state transformers read the action trace of a thread:

1. If the state transformers compute the state for the absolute point in time t, they read the current action of the highest prioritized ready thread τ_h that is at the head of the FIFO list of this priority. This action is **action(t)**. It denotes what τ_h will do in between t and $t+1$;

2. If a blocked thread τ_b unblocks immediately at t or if an inactive thread is released immediately at t, the current action of τ_b is read to determine in which thread state τ_b is unblocked. Notice, a thread can be released in the blocked state (e.g., if its previous job has stopped while awaiting the reception of a message); and

3. If the scheduler deactivates *Countermeasure II* after having delayed a thread τ_b, the current action of τ is read to determine the thread state of τ.

The current actions of other threads are ignored. **action(t)** denotes the action a thread will do in between t and $t + 1$. Actions that last longer than one unit of time (i.e., longer than this time) are encoded by repeating the same action at successive points in time in the action trace. The above formalization simplifies the verification in two ways:

1. Action traces are constant parameters. This relieves the state transformers form maintaining a list of remaining actions in **Dynamic_State**.

2. Because action traces are a parameter in **Constant_Secrecy_Dependent_State**, l-similarity can simply require these parameters to agree for l-observable threads τ (i.e., for threads for which $dom(\tau) \leq l$ holds).

The eight state transformers take the variable **sts** as an input parameter. The type of this variable is: **[Thread → Constant_Secrecy_Dependent_State]**.

3.4.3. State Transformers

The following eight state transformers formalize the behavior of the scheduler.

```
% State transformers
stop_delaying_thread(sts)(s, time)   : State = ...
deactivate_thread(sts)(s, time)      : State = ...
release_thread(sts)(s, time)         : State = ...
unlock_blocked_thread(sts)(s, time)  : State = ...
block_thread(sts)(s, time)           : State = ...
resume_delayed_thread(sts)(s, time)  : State = ...
delay_preemptions(sts)(s, time)      : State = ...
run(sts)(s, time)                    : State = ...
```

The input parameters of these state transformers are **s**, **time**, and **sts**. The PVS constant **cts** is a further implicit constant. The parameter **time** of type **Time** denotes the absolute point in time that corresponds to the state **s**. Taken together, these eight state transformers produce a state that corresponds to **time + 1**. The parameters **s**, **sts**, and **cts** contain the dynamic thread state, the secrecy dependent state, and the secrecy independent state of all threads.
The wrapper **dispatch_step** combines all eight state transformer in the order listed above. That is, the result of **stop_delaying_thread** is passed as input to **deactivate_thread**, and so on. The wrapper **dispatch** takes an initial state s_0 and a time $time_{max}$ and invokes **dispatch_step** recursively for all points in time $0 \leq time \leq time_{max}$.

The PVS encoding of **dispatch_step** and of **dispatch** is:

```
dispatch_step(sts)(s, time) : State =
  run(sts)(
    delay_preemptions(sts)(
      resume_delayed_thread(sts)(
        block_thread(sts)(
          unblock_blocked_thread(sts)(
            release_thread(sts)(
              deactivate_thread(sts)(
                stop_delaying_thread(sts)(s, time)
              , time)
            , time)
          , time)
        , time)
      , time)
    , time)
  , time)
```

and:

```
dispatch(sts)(s, time_max) : Recursive State =
  If time_max = 0 Then
    dispatch_step(sts)(s, 0)
  Else
    dispatch_step(sts)(dispatch(sts)(s, time_max − 1), time_max)
  Endif
 Measure time_max
```

Before I present the formalization of the above eight state transformers, let me introduce the helper functions I used in their formalization.

3.4.3.1. Helper Functions

The formalization of the eight state transformers makes use of the following two helper functions to identify the highest prioritized thread that executes non-preemptively and that is at the head of the FIFO list respectively the highest prioritized treated-as-ready thread that is also at the head of this list. They are defined as:

```
highest_prioritized_non_preemptively_executing_thread(sts)(s) : Thread =
  singleton_elt (λ ( t_l : Thread) :
    s( t_l )‘remaining_max_delay > 0 ∧
      Forall (t_h : Thread) :
        higher_effective_priority (sts)(s)( t_l , t_h) ⟹ ¬ s(t_h)‘remaining_max_delay > 0)
```

and as:

```
highest_prioritized_treated_as_ready (sts)(s) : Thread =
  singleton_elt (λ (t : Thread) :
    treated_as_ready(s)(t) ∧
      Forall (t_h : Thread) : higher_effective_priority (sts)(s)(t, t_h) ⟹
        ¬ treated_as_ready(s)(t_h))
```

where **s** is a state in which a non-preemptively executing thread respectively in which a treated-as-ready thread must exists. The above two functions are undefined if no such thread exist [9].

In the above two functions, **singleton_elt(S)** returns the single element of the singleton set **S** [10]. The relation **higher_effective_priority(sts)(s)** establishes a strict total order

[9] In the PVS sources [Völ10], this requirement is expressed with a suitable predicate subtype of **State**.

[10] A singleton set is a set, which contains precisely one element.

between threads. A thread **t_h** has a higher effective priority than a thread **t_l** (i.e., **higher_effective_priority(sts)(s)(t_l, t_h)** holds) if either **t_h** has a higher priority or if both threads have the same priority and **t_h** precedes **t_l** in the FIFO list. This is the case if the last release point of **t_h** was earlier than the last release point of **t_l**. A clash of equally prioritized simultaneously released threads is resolved with an arbitrary secrecy independent total order on threads (e.g., smaller values of a large random number that is stored in the thread control block at thread creation time). Here we use the natural numbers **t_h** and **t_l**. The PVS encoding of **higher_effective_priority** is straightforward

```
higher_effective_priority (sts)(s)( t_l , t_h : Thread) : bool =
 % t_h is higher prioritized than t_l
 (cts( t_l )‘ prio < cts(t_h )‘ prio ) ∨
 % t_h and t_l share the same priority, t_h is released earlier
 (cts( t_l )‘ prio = cts(t_h )‘ prio ∧
    sts(t_h )‘ release_point(s(t_h )‘ job) < sts( t_l )‘ release_point(s( t_l )‘ job)) ∨
 % t_h < t_l to avoid clashes of equally prioritized simultaneously released threads
 (cts( t_l )‘ prio = cts(t_h )‘ prio ∧
    sts(t_h )‘ release_point(s(t_h )‘ job) = sts( t_l )‘ release_point(s( t_l )‘ job) ∧
      t_h < t_l )
```

The predicate over threads **treated_as_ready** returns true for all threads that are ready or that are treated-as-ready. That is, it holds for a thread **t** either if **t** is ready or delayed, or if the countermeasure predicate $p_{transitive}$ holds for this thread and it is either blocked or it has stopped. Recall, $p_{transitive}$ is the countermeasure predicate for *Countermeasure I*, which avoids leakage due to direct and indirect influence (see Definition 11 on page 64). The PVS code for **treated_as_ready** is:

```
treated_as_ready(s)(t : Thread) : bool =
  is (Ready)(s, t) ∨ is (Delayed)(s, t) ∨
    p_transitive (t) ∧ (is (Blocked)(s, t) ∨ is (Stopped)(s, t))
```

In **treated_as_ready** (and elsewhere), **is(Ready)(s, t)** abbreviates **s(t)‘thread_state = Ready**. The predicate, **is_delayed(sts)(s) : bool** returns true if there exists a thread in s that executes non-preemptively.

Two further helper functions are: **apply_action** and **unblock_thread** [11]. The function **apply_action** reads the current action from the action trace of a given thread **t** and updates the member **thread_state** of **s(t)** accordingly. If **t** intends to run (preemptively or non-preemptively), it becomes **Ready**. A thread that stops executing or that blocks becomes **Stopped** and **Blocked**, respectively.

```
apply_action(sts)(s, time, t) : Dynamic_State =
  Cases sts(t)‘action(time) Of
    run                  : s(t) With [(thread_state) := Ready],
    run_non_preemptively : s(t) With [(thread_state) := Ready],
    stop                 : s(t) With [(thread_state) := Stopped],
    block                : s(t) With [(thread_state) := Blocked]
  EndCases
```

The second helper function **unblock_thread** wraps **apply_action** to update the dynamic state of a thread that unblocks or that is released. In situations where the scheduler has to apply *Countermeasure II* to avoid timing channels due to non-preemptive execution, **unblock_thread** sets

[11] Recall from Section 2.7: the notion **x With [(y) := z]** stands for a partial update of the record variable **x**. The member **y** of this record is set to the value **z**.

the thread to **Delayed** and initializes the clock **effective_release** to **max_delay_low**. The countermeasure predicate p_{delay} holds in this case (see Definition 15 on page 70). In situations where the currently running thread executes non-preemptively, the unblocked thread is also set to **Delayed** however without setting **effective_release**. In this case, the thread resumes executing immediately when the lower prioritized thread stops executing non-preemptively. Threads at a lower priority than the non-preemptively executing thread are not affected. The function **unblock_thread** can therefore directly invoke **apply_action** without first having to set the **thread_state** to **Delayed**. **apply_action** is also invoked to unblock a thread immediately if no thread executes non-preemptively and if *Countermeasure II* needs not to be applied for the unblocking thread. The PVS code of **unblock_thread** is:

```
unblock_thread(sts)(s, time, t) : Dynamic_State =
  If p_delay(t)
  Then
    s(t) With [(thread_state)        := Delayed,
               (effective_release)  := cts(t)`max_delay_low]
  Else
    If is_delayed(sts)(s) ∧
       cts(highest_prioritized_non_preemptively_executing_thread(sts)(s))`prio ≤ cts(t)`prio
    Then
       s(t) With [(thread_state)    := Delayed]
    Else
       apply_action(sts)(s, time, t)
    Endif
  Endif
```

3.4.3.2. Stop Delaying Thread

The first of the eight state transformers is **stop_delaying_thread**. It resets **s(dt)`remaining_max_delay** if the non-preemptively executing thread (**dt**) re-enables preemptions or if it has contiguously executed non-preemptively for $max\_delay_i$. The former is the case if its current action is not **run_non_preemptively**. The latter is the case if **s(dt)`remaining_max_delay = 0**. The PVS code of **stop_delaying_thread** is:

```
stop_delaying_thread(sts)(s, time) : State =
  If is_delayed(sts)(s)
  Then
    Let dt = highest_prioritized_non_preemptively_executing_thread(sts)(s) In
      If
        (s(dt)`remaining_max_delay = 0 ∨
         ¬ run_non_preemptively(sts(dt)`action(time)))
      Then
        s With [(dt`remaining_max_delay) := 0]
      Else
        s
      Endif
  Else
    s
  Endif
```

In those L4-family microkernels that implement the `delayed preemption` mechanism, the scheduling decision, which is formalized in **stop_delaying_thread**, is triggered in two situations:

- if the kernel-programmed timeout fires after max_delay_i; or
- if the thread voluntarily yields the CPU (e.g., after it has seen that a preemption is pending).

3.4.3.3. Deactivate Thread

The second state transformer is **deactivate_thread**. It sets a thread to inactive either if the total budget of this thread is depleted or if its deadline has passed. Because non-preemptive execution cannot delay the deactivation due to total budget depletion or due to a passing deadline, **deactivate_thread** needs not to consider non-preemptively executing threads. The PVS code of this state transformer is:

```
deactivate_thread(sts)(s, time) : State =
  λ (t : Thread) :
    If s(t)`remaining_total_budget = 0 ∨
       s(t)`remaining_deadline = 0
    Then
      s(t) With [(thread_state)           := Inactive,
                 (job)                    := s(t)`job + 1,
                 (remaining_deadline)     := 0,
                 (remaining_total_budget) := 0]
    Else
      s(t)
    Endif
```

In an implementation of the scheduler, the scheduling decision, which **deactivate_thread** describes, is triggered by a timeout. The kernel programs this timeout immediately before it returns control to this thread. It sets the timeout to the minimum of the remaining total budget, of the remaining execution budget, and of the time that remains until the thread's deadline passes. If a thread is blocked or if it has stopped, the deactivation can occur lazily (e.g., as described in Footnote 3 on page 50).

3.4.3.4. Release Thread

The third state transformer — **release_thread** — releases a thread whose next release point has occurred. For that, it invokes the helper function **unblock_thread** and refills the remaining total budget of this thread to the total budget of its released job $\tau_{i,k}$. Moreover, it resets the remaining time until the deadline expires to $d_{i,k}$. The PVS code is:

```
release_thread(sts)(s, time) : State =
  λ (t : Thread) :
    If time = sts(t)`release_point(s(t)`job) ∧
       is(Inactive)(s,t)
    Then
        unblock_thread(sts)(s, time, t)
                          With [(remaining_total_budget) := sts(t)`total_budget(s(t)`job),
                                (remaining_deadline)     := sts(t)`deadline(s(t)`job)]
    Else
      s(t)
    Endif
```

In an implementation of the scheduler, a release is triggered by a timeout, by the occurrence of a releasing event or by both.

3.4.3.5. Unblock Blocked Thread

A blocked thread resumes its execution if its next action is not **block**. In this case, the fourth state transformer — **unblock_blocked_thread** — resumes this thread by invoking the helper function **unblock_thread**.

```
unblock_blocked_thread(sts)(s, time) : State =
  λ (t : Thread) :
    If is(Blocked)(s, t) ∧
       ¬ block(sts(t)‘action(time))
    Then
       unblock_thread(sts)(s, time, t)
    Else
       s(t)
    Endif
```

In an implementation of the scheduler, unblocking is typically triggered by an unblocking event such as the reception of a message or the occurrence of an interrupt.

3.4.3.6. Resume Delayed Thread

The fifth state transformer — **resume_delayed_thread** — resumes the normal execution of a thread that is in the state **Delayed**. The normal execution of a thread τ is resumed after **effective_release** reaches zero. This is the case either after *Countermeasure II* was active for $max\_delay\_low(\tau)$ (see Definition 14 on page 14) or after the currently running thread has stopped executing non-preemptively.

```
resume_delayed_thread(sts)(s, time) : State =
  If (Exists (t : Thread) : treated_as_ready(s)(t)) ∧
     ¬ is_delayed(sts)(s)
  Then
    Let t = highest_prioritized_treated_as_ready (sts)(s) In
      If is(Delayed)(s, t) ∧
         s(t)‘ effective_release = 0
      Then
        s With [(t) := apply_action(sts)(s, time, t)]
      Else
        s
      Endif
  Else
    s
  Endif
```

The pre-located trampoline code, which I have described in Section 3.3.5.2 on page 73 is one implementation of this state transformer.

3.4.3.7. Block Thread

The sixth state transformer — **block_thread** — formalizes the state changes that occur when the currently running thread blocks or when it stops. In these situations, it sets the thread's **thread_state** to **Blocked** respectively to **Stopped**.

```
block_thread(sts)(s, time) : State =
  If Exists (t : Thread) : treated_as_ready(s)(t)
  Then
    Let t = highest_prioritized_treated_as_ready (sts)(s) In
      If is(Ready)(s, t) ∧
         ¬ s(t)‘remaining_max_delay > 0
      Then
        If block(sts(t)‘action(time)) Then
          s With [(t) := s(t) With [(thread_state) := Blocked]]
        Else
          If stop(sts(t)‘action(time)) Then
            s With [(t) := s(t) With [(thread_state) := Stopped]]
          Else
            s
          Endif
        Endif
      Else
        s
      Endif
  Else
    s
  Endif
```

3.4.3.8. Delay Preemptions

The seventh state transformer — **delay_preemption** — formalizes the effect of a non-preemptive execution of the currently running thread. The PVS code for **delay_preemption** is:

```
delay_preemptions(sts)(s, time) : State =
  If Exists (t : Thread) : treated_as_ready(s)(t)
  Then
    Let t = highest_prioritized_treated_as_ready (sts)(s) In
      If is(Ready)(s, t) ∧
         ¬ s(t)‘remaining_max_delay > 0 ∧
         run_non_preemptively(sts(t)‘action(time)) ∧
         cts(t)‘max_delay > 0 ∧
         cts(t)‘max_delay ≤s(t)‘remaining_total_budget ∧
         cts(t)‘max_delay ≤s(t)‘remaining_deadline
      Then
        s With [(t) := s(t) With [(remaining_max_delay) := cts(t)‘max_delay]]
      Else
        s
      Endif
  Else
    s
  Endif
```

The proof of non-interference also holds for systems where some threads cannot execute non-preemptively. In the formal model of the scheduler, these threads are represented by $max\_delay_i = 0$. The check $max\_delay_i > 0$ determines whether a thread is authorized to execute non-preemptively. The check

```
cts(t)‘max_delay ≤s(t)‘remaining_total_budget ∧
  cts(t)‘max_delay ≤s(t)‘remaining_deadline
```

disallows non-preemptive execution during end-of-release events.

In L4 kernels, the intent of an application thread to execute non-preemptively is only detected at the occurrence of an interrupt. If the in-kernel interrupt handler notices that the `delayed preemption` mechanism is active, it returns control to the application program. In the formal model of the scheduler, this situation is represented by action traces, which contain **run** until the point in time when the kernel would notice the intent to execute non-preemptively and which then contains **run_non_preemptively** to express precisely this intent.

3.4.3.9. Run

The eighth state transformer — **run** — formalizes no scheduling decision.

```
run(sts)(s, time) : State =
  λ (t : Thread) :
    s(t) With [(effective_release)   := If s(t)‘effective_release > 0 Then
                                           s(t)‘effective_release − 1
                                        Else s(t)‘effective_release Endif,
               (remaining_max_delay)     := If s(t)‘remaining_max_delay > 0 Then
                                           s(t)‘remaining_max_delay − 1
                                        Else 0 Endif,
               (remaining_deadline) := If s(t)‘remaining_deadline > 0 Then
                                           s(t)‘remaining_deadline − 1
                                        Else s(t)‘remaining_deadline Endif,
               (remaining_total_budget)    := If highest_prioritized_treated_as_ready(sts)(s)(t) ∧
                                           s(t)‘remaining_total_budget > 0 Then
                                           s(t)‘remaining_total_budget −1
                                        Else s(t)‘remaining_total_budget Endif]
```

The sole purpose of the state transformer **run** is to advance the various down-counting clocks, which I use to emulate timeouts. The total budget of a thread is only consumed if this thread is the highest prioritized treated-as-ready thread. Hence, **remaining_total_budget** is a thread-local clock.

3.4.4. Invariants

To give some confidence on the correctness of this formalization, I have shown the following six predicates over states to be invariants. The formalization of the first four predicates are straightforward. I will therefore explain these predicates only informally. A predicate P is invariant of the scheduler if $P(dispatch\_step(s_0, 0))$ holds for an initial state s_0 and if for all points in time t, we can conclude from $P(dispatch(s_0, t))$ that $P(dispatch(s_0, t + 1))$ holds as well.

- **delay_max**
  cts(t)‘max_delay is an upper bound of **s(t)‘remaining_max_delay**.

- **delaying_max_delay**
  Only threads τ_i with $max\_delay_i > 0$ can execute non-preemptively.

- **delay_remaining**
  No thread exceeds its remaining total budget or its deadline when it delays preemptions. The proof that **delay_remaining** is an invariant of the scheduler rests on **delay_max**.

- λ **(s) : singleton_delaying(s)** $\wedge$ **delaying_ready(s)**
  The first clause of this predicate states that at most one thread is delaying preemptions. The second clause establishes that the delaying thread is **Ready** and that no higher prioritized thread is in this state. These threads are therefore either **Delayed**, **Blocked**, **Stopped** or **Inactive**. The proof that this predicate is an invariant of the scheduler rests on **delay_remaining**.

The predicates of the fifth and sixth invariant are:

```
delay_effective_release (sts)(s) : bool =
  Forall ( t_l , t_h : Thread) :
    s( t_l )‘ remaining_max_delay > 0 ∧ s(t_h)‘thread_state = Delayed ∧
      higher_effective_priority (sts)(s)( t_l , t_h) ∧ p_delay(t_h)
  ⟹
    s( t_l )‘ remaining_max_delay ≤s(t_h)‘ effective_release
```

and

```
p_delay_delayed(sts)(s) : bool =
  Forall ( t_l , t_h) :
    s( t_l )‘ remaining_max_delay > 0 ∧ higher_effective_priority (sts)(s)( t_l , t_h) ∧
    treated_as_ready(s)(t_h) ∧ p_delay(t_h)
  ⟹
    s(t_h)‘ thread_state = Delayed
```

The fifth predicate states that a delayed higher prioritized thread τ, which is subjected to *Countermeasure II* (i.e., for which $p_{delay}(\tau)$ holds) resumes execution only after a lower prioritized thread stopped executing non-preemptively. The proof that this predicate is a scheduler invariant rests on **delay_max** and **delay_remaining**.

The sixth invariant establishes that whenever a thread executes non-preemptively, higher prioritized treated-as-ready threads that the scheduler subjects to *Countermeasure II* are **Delayed**. The proof rests on **singleton_delaying** and **delay_remaining**.

The proofs of the above six invariants are straightforward by case distinction.

3.4.5. Non-interference

The non-interference property that I have shown for the proposed budget-enforcing fixed-priority scheduler establishes the indistinguishability of outputs on l-similar initial scheduler states. Its PVS code is the following.

```
main_theorem : Theorem
  Forall (s_0 : ( initial_state ), t_conf : Thread, time : Time,
          sts1, sts2 : [Thread → Constant_Secrecy_Dependent_State]) :
    l_similar (t_conf )(sts1, sts2) ⟹
      output(cts(t_conf )‘ label, sts1)(dispatch(sts1)(s_0, time)) =
      output(cts(t_conf )‘ label, sts2)(dispatch(sts2)(s_0, time))
```

In the definition of this theorem, the predicate subtype **(initial_state)** stands for the initial dynamic state in which all threads are inactive awaiting the release point of their first job. The observer clearance l is given as the domain of the thread **t_conf**.

Two constant secrecy-dependent states **sts1** and **sts2** are l-similar if the following relation contains the pair **(sts1, sts2)**.

```
l_similar (t_conf : Thread)(sts1, sts2) : bool =
  (Forall (t_low : Thread) :
     cts(t_low)‘release_label ≤ cts(t_conf)‘label ⟹ same_params(t_low)(sts1, sts2)) ∧
  (Forall (t_low : Thread) :
     cts(t_low)‘label ≤ cts(t_conf)‘label ⟹
       sts1(t_low)‘action = sts2(t_low)‘action)
```

In this relation, **same_params** is defined as:

```
same_params(t)(sts1, sts2) : bool =
       (sts1(t)‘release_point = sts2(t)‘release_point) ∧
       (sts1(t)‘deadline      = sts2(t)‘deadline) ∧
       (sts1(t)‘total_budget  = sts2(t)‘total_budget)
```

If an l-classified observer (represented by **t_conf**) is cleared to observe the existence of a thread τ, then two constant secrecy-dependent states are l-similar if they agree on the release points, deadlines and total budgets of this thread. If the l-classified observer is also cleared to receive information from τ, the action trace of this thread must also be the same for the two states to be l-similar.

The PVS code for **output** is:

```
output(l, sts)(s) : Output =
  If is_delayed(sts)(s) Then
    Let dt = highest_prioritized_non_preemptively_executing_thread(sts)(s) In
      If cts(dt)‘label ≤ l Then
        delay_out(dt)
      Else
        nil
      Endif
  Else
    If (Exists (t : Thread) : treated_as_ready(s)(t)) ∧
        s(highest_prioritized_treated_as_ready(sts)(s))‘thread_state ≠ Delayed
    Then
      Let ht = highest_prioritized_treated_as_ready(sts)(s) In
        If cts(ht)‘label ≤ l Then
          action(ht, s(ht)‘thread_state)
        Else
          nil
        Endif
    Else
      nil
    Endif
  Endif
```

The function **output** maps each state of the scheduler to an element of the type:

```
Output : Datatype
Begin
  nil : nil?
  delay_out (t : Thread)                  : delay_out?
  action    (t : Thread, ts : Thread_State) : action?
End Output
```

It contains three variants:

- **nil** — the observer is not cleared to the currently running thread;
- **delay_out(t)** — the currently running thread is **t**. It executes non-preemptively; and
- **action(t, ts)** — the highest prioritized treated-as-ready thread is **t**. Its thread state is **ts**.

What do we know about a scheduler for which the main theorem holds? If such a scheduler starts from two l-similar initial scheduler states, which are comprised of **s_0**, **cts** and **sts1** respectively of **s_0**, **cts** and **sts2**, then l-classified observers cannot distinguish the two schedules from the behavior l-observable threads have at a given point in time. Quantification over the parameter **time**, which is passed to **dispatch**, extends this statement to all points in time and to arbitrarily long running systems. Thereby, initial scheduler states can differ in the actions higher or incomparably classified threads will perform and on the existence of threads that run at a lower priority than the lowest prioritized l-observable thread.

The non-interference property of the main theorem, as states above, is a simplified form of non-influence: Definition 6 on page 24 introduced non-influence as

$$\begin{aligned} &\forall \alpha, \beta \in A^*, s^0, s^i, t^0 \in S.\ \mathit{ipurge}(\alpha, l) = \mathit{ipurge}(\beta, l)\ \wedge\ s^0 \overset{\mathit{sources}(\alpha,l)}{\approx} t^0\ \wedge\ s^0 \xrightarrow{\alpha} s^i \\ &\qquad \Rightarrow \quad \exists t^j \in S.\ t^0 \xrightarrow{\beta} t^j \wedge \mathit{output}(l, s^i) = \mathit{output}(l, t^j) \end{aligned} \tag{3.5}$$

The encoding of actions as mappings of type **[Time $\rightarrow$ Action]** and the transitivity of $\leq$ simplifies the precondition $\mathit{ipurge}(l, \alpha) = \mathit{ipurge}(l, \beta)$ to **sts1(t)‘action = sts2(t)‘action** for all lower-than-l classified threads **t**.
$s \overset{\mathit{sources}(\alpha,l)}{\approx} t$ is simplified to the first condition of **l_similar(t_conf)(sts1, sts2)**. For the dynamic part of the state $s \overset{\mathit{sources}(\alpha,l)}{\approx} t$ holds trivially because in the above lemma, dispatch starts from the same initial state. Also, because **dispatch(s_0, time)** is deterministic and because it terminates after **time** steps, the quantification over s^i and the existence of t^j can simply be expressed as the result of **dispatch(sts1)(s_0, time)** respectively as the result of **dispatch(sts2)(s_0, time)**. The function output of Definition 6 is instantiated with the above output function.

3.4.6. Proof of Non-interference

The non-interference proof of the proposed budget-enforcing scheduler proceeds in two steps: First, it must be shown that the property on pairs of states — **same_high_state** — is invariant for the given scheduler. Then, the main theorem follows immediately from the following lemma.

```
same_high_state_same_output : Lemma
  Forall (s1 : (scheduler_invariants(sts1)), s2 : (scheduler_invariants(sts2))) :
    l_similar (t_conf)(sts1, sts2) ∧ same_high_state(t_conf)(s1, s2)
  ⟹
    output(cts(t_conf)‘label, sts1)(s1) = output(cts(t_conf)‘label, sts2)(s2)
```

It says that two states, which are related by **same_high_state**, yield identical outputs as seen by an l-classified observer.

The predicate **same_high_state** is defined as follows.

```
same_high_state(t_conf)(s1, s2 : State) : bool =
  (Forall (t) : cts(t)`label ≤cts(t_conf)`label ⟹ s1(t) = s2(t)) ∧
  (Forall (t_l , t_h) :
     cts( t_l )`label ≤cts(t_conf)`label ∧
     cts( t_l )`prio ≤cts(t_h)`prio ⟹
       (s1(t_h)`job                          = s2(t_h)`job ∧
        s1(t_h)`remaining_total_budget       = s2(t_h)`remaining_total_budget ∧
        s1(t_h)`remaining_deadline           = s2(t_h)`remaining_deadline ∧
        (s1(t_h)`thread_state = Inactive) = (s2(t_h)`thread_state = Inactive)))
```

The first clause of this predicate says that if an l-classified observer (represented by **t_conf**) is cleared to receive information about a thread **t**, then the dynamic state of this thread is the same in both states **s1** and **s2** of the related pair. Because the property is an invariant of the scheduler, it follows for l-similar initial states that the behavior of l-observable threads cannot be altered by higher or incomparably classified threads.

The second clause of this predicate says that for threads, which have the same or a higher priority than an l-observable thread, the following holds: they execute the same jobs in the two states of the pair; they have the same amount of total budget left to execute or to block in the subsequent schedule that follows the two state of the pair; their deadlines expire after the same amount of time; and the jobs that these threads execute are either active in both states or they are inactive in both states. That is, in the two schedules, higher prioritized threads are either simultaneously active or they are simultaneously inactive. If an l-observable lower or equally prioritized thread is not cleared to receive information from these threads, we can conclude from this second clause that *Countermeasure I* is applied for such a thread at the same time in both of these schedules.

The proof of **same_high_state_same_output** is by distinction of the cases that stem from the if-statements in the function **output**. The cases where the function **output** evaluates to **nil** for both states **s1 = dispatch(sts1)(s_0, time)** and **s2 = dispatch(sts1)(s_0, time)** hold trivially.

Of the remaining four cases, the following two are straightforward:

Case 1: in both states, **s1** and **s2**, there exists a highest prioritized thread that executes non-preemptively and the l-classified observer is cleared to see these threads; and

Case 2: a highest prioritized treated-as-ready thread exists in both **s1** and **s2** and the l-classified observer is authorized to see these threads.

The remaining two cases are more challenging:

Case 3: the condition **is_delayed** holds in precisely one of the two states, **s1** and **s2**, and the l-classified observer is not cleared to see the highest prioritized non-preemptively executing thread; and

Case 4: a highest prioritized non-preemptively executing thread respectively a highest prioritized treated-as-ready thread exists in both, **s1** and **s2**, but the l-classified observer is cleared only to one of these threads.

The proofs of Case 1 and Case 2 proceed by instantiating the parameter **t** in the precondition **cts(t)`label ≤cts(t_conf)`label** of the first clause of **same_high_state(t_conf)(s1, s2)** with the highest prioritized non-preemptively executing thread (Case 1) respectively with the highest prioritized treated-as-ready thread (Case 2) of one of these states. Because **cts(t_conf)‘label = l**, **cts(t)‘label** $\leq$ **cts(t_conf)‘label** holds for both of these threads and from the first clause of **same_high_state(t_conf)(s1, s2)** it follows that **s1(t) = s2(t)**. However then, the singleton delaying thread in **s1** is the same thread as the singleton delaying thread in **s2** and the singleton highest prioritized treated-as-ready thread in **s1** is the singleton highest prioritized treated-as-ready thread in **s2**. But this means, the l-observable outputs are the same. In the sources, two auxiliary lemmas establish this identity of the above threads:

- **visible_high_priority_runnable_thread_same**, and
- **visible_high_priority_delaying_thread_same**.

For the proof of Case 3, let us assume that **is_delayed** holds in **s1** but not in **s2**. The proof where **is_delayed** holds in **s2** is symmetric. Because the l-classified observer is not cleared to see the highest prioritized non-preemptively executing thread that exists in **s1**, **output(sts1)(s1) = nil** holds. The function **output(sts2)(s2)** can match this result only in the following two situations:

- if there is no highest prioritized treated-as-ready thread that is not **Delayed**, or
- if the l-classified observer is not cleared to see such a thread.

So let us assume that there exists such a thread in **s2**, which the l-classified observer is cleared to see. Our goal is now to find a contradiction. Because of the above assumption, the precondition of the first clause of **same_high_state(t_conf)(s1, s2)** is fulfilled. From **visible_high_priority_runnable_thread_same** we know that in this case, the highest prioritized treated-as-ready thread in **s2** is also the highest prioritized treated-as-ready thread in **s1**. Let τ_h be this thread. We have to distinguish two cases:

Case 3a: *The singleton non-preemptively executing thread τ_d is authorized to send to the highest prioritized treated-as-ready thread τ_h.*

In this case, the transitivity of $\leq$ immediately reveals that τ_d is also authorized to send to the l-classified observer, which contradicts the second precondition of Case 3.

Case 3b: *No such communication is authorized.*

In this case, the countermeasure predicate $p_{delay}(\tau_h)$ does not hold for the highest prioritized treated-as-ready thread τ_h because otherwise the invariant **p_delay_delayed** would require this thread to be **Delayed**.

However, the non-preemptively executing thread τ_d has the same or a lower priority than τ_h and **max_delay** $>$ **0** holds for τ_d. But then, $p_{delay}(\tau_h)$ must hold, which contradicts the assumption that τ_h is not **Delayed**.

Case 4 contradicts the results of **visible_high_priority_delaying_thread_same** respectively of **visible_high_priority_runnable_thread_same**. If the l-classified observer is cleared to see such a thread in one of the two states **s1** or **s2**, this thread is the single highest prioritized non-preemptively executing respectively the single highest prioritized treated-as-ready thread in the respective other state. As a consequence, the l-classified observer is cleared to see this thread. This concludes the proof of **same_high_state_same_output**.

The proofs of the lemmas, which establish that **same_high_state** is a scheduler invariant for the eight state transformers, are straightforward. The accompanying PVS sources contain these proofs.

3.4.7. Temporal Isolation of Non-interfering Threads

From the above proof, we can immediately conclude that the proposed budget-enforcing fixed-priority scheduler isolates threads in a temporal manner from certain other threads: Assume $dom(\tau) \not\leq dom(\tau')$ holds for two threads τ and τ' and $(L, \leq, dom)$ is a transitive information-flow policy. Then, τ can neither influence the points in time when τ' runs nor the points in time when a thread τ'' runs that can legitimately send messages to τ'. The subsystem, which consists of τ' and of all its legitimate senders τ'', is temporally isolated from τ.

Commercial time-partitioning systems (such as LynxOS [Lyn]) often implement a hierarchical fixed-priority scheduler to schedule the threads of a single partition once the underlying scheduler selects this partition. These systems can only temporally isolate threads by running them in different partitions. The proposed budget-enforcing fixed-priority scheduler directly isolates threads in a temporal manner without having to revert to hierarchical scheduling.

3.5. Real-Time Guarantees

In real-time systems, it is crucial that hard real-time threads complete their jobs before their deadlines. Admission tests give this guarantee. Probably the two most popular admission tests are the *time-demand analysis* by Lehoczky et al. [LSD89] and the Liu and Layland criterion [LL73] for the rate-monotonic scheduling (RMS) algorithm.

In the following, we shall see how an adjustment of the per thread blocking term allows us to reuse a large class of existing admission tests for the proposed budget-enforcing fixed-priority scheduler. We shall further see how the above two admission tests perform in comparison to their adjusted versions and how the latter perform in comparison to admission tests for time-partitioning schedulers.

3.5.1. Time-Demand Analysis and Liu-Layland Criterion

Time demand analysis delivers sufficient and necessary conditions to test whether a given set of threads is schedulable, that is, whether all threads in this set meet their deadlines. If the relative deadlines of all jobs $\tau_{i,k}$ are at most as large as the period of the thread (i.e., $d_{i,j} \leq \Pi_i$), the time demand to schedule a thread τ_i in the interval $[t_0,\ t_0 + t]$ along with higher prioritized threads [12] is

$$w_i(t) = eb_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{\Pi_k} \right\rceil eb_k, \quad \mathit{for}\ 0 < t \leq \Pi_i \tag{3.6}$$

Here, eb_i is the worst case time τ_i can execute (i.e., its execution budget), the time t_0 is a critical instant for τ_i. That is, t_0 is the release point a job of τ_i where the response time of this job will be at its maximum. If a job is schedulable at a critical instant, it remains schedulable for all other combinations of release times (see e.g., Liu [Liu00, Chapter 6.5.1] for more details on critical instant analyses).

A job of τ_i meets its deadline if at some time t before its deadline, the supply of processor time t is equal to or greater than the time demand $w_i(t)$.

The Liu Layland criterion is a sufficient condition for the schedulability of a set of strictly periodic threads that are scheduled by the rate-monotonic scheduling algorithm. Recall, the

[12] In the following, thread indices are assigned inverse proportional to thread priorities. That is, threads with smaller indices are higher prioritized: $i < j \Rightarrow prio(\tau_i) \geq prio(\tau_j)$.

RMS algorithm prioritizes threads inverse proportionally to their period lengths (i.e., $\Pi_i \leq \Pi_j \Rightarrow prio(\tau_i) > prio(\tau_j)$). A set of n strictly periodic threads is schedulable if

$$\sum_{i=1}^{n} \frac{eb_i}{\Pi_i} \leq n \cdot (2^{\frac{1}{n}} - 1) \tag{3.7}$$

A limitation of the above two schedulability tests is the implicit assumption that threads never block. To consider the blocking of threads, extended versions of these admission tests include a per thread blocking term bb_i. This blocking term is an upper bound of the time that τ_i blocks. Sha et al. [SRL90] show that blocked threads remain schedulable with RMS if Equation 3.8 holds.

$$\forall k \in \{1, ...n\}. \sum_{i=1}^{k} \left(\frac{eb_i}{\Pi_i} + \frac{bb_k}{\Pi_k} \right) \leq k \cdot (2^{\frac{1}{k}} - 1) \tag{3.8}$$

Equation 3.9 shows an analogous adjustment for time demands.

$$w_i(t) = eb_i + bb_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{\Pi_k} \right\rceil eb_k, \quad for\ 0 < t \leq \Pi_i \tag{3.9}$$

If the effect of *Countermeasure I* (see Definition 10 on page 61) on other threads can be expressed as a blocking term, admission tests such as the two above could immediately be reused to decide whether a set of threads is schedulable with the proposed budget-enforcing fixed-priority scheduler. The *prohibition times* bb^{pr} is such a blocking term.

3.5.2. Prohibition Times

In situations where the scheduler activates *Countermeasure I* to avoid leakage due to direct and indirect influences, only budget-consumer threads can run while the highest prioritized treated-as-ready thread blocks or stops.

Let bb_h be the maximum time that τ_h can block without consuming its execution budget (i.e. $bb_h = tb_h - eb_h$). In the worst case, the scheduler switches to a budget consumer whenever a higher prioritized thread blocks for which $p_{transitive}(\tau_h)$ holds. A lower prioritized thread τ_l is prohibited from running. I call this time bb_l^{pr} the *prohibition time* of a thread τ_l. It holds:

Proposition 1. Prohibition Times.
For a thread τ_l of the set of threads T, the prohibition time bb_l^{pr} is:

$$bb_l^{pr} = \sum_{\tau_h \in T_{H+}} \left\lceil \frac{\Pi_l}{\Pi_h} \right\rceil .bb_h \quad \textbf{with } T_{H+} := \{\tau \in T_{high}(\tau_l) \mid p_{transitive}(\tau)\}$$

Obviously, if the idle thread is the budget consumer or if no other higher classified, ready budget consumer could be found, prohibiting threads from running increases the idle time of the system. I quantify the worst-case increase of idle time by the prohibition time of the lowest prioritized thread bb_{idle}^{pr}.

To reuse existing admission tests for the proposed scheduler, the prohibition time must be considered as an additional blocking term. In addition, the influence of threads τ with $p_{transitive}(\tau)$ must be removed form the original blocking term of a thread τ_i. For the time-demand analysis, this results in:

$$w_i(t) = eb_i + bb_i' + bb_i^{pr} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{\Pi_k} \right\rceil eb_k, \quad for\ 0 < t \leq \Pi_i \tag{3.10}$$

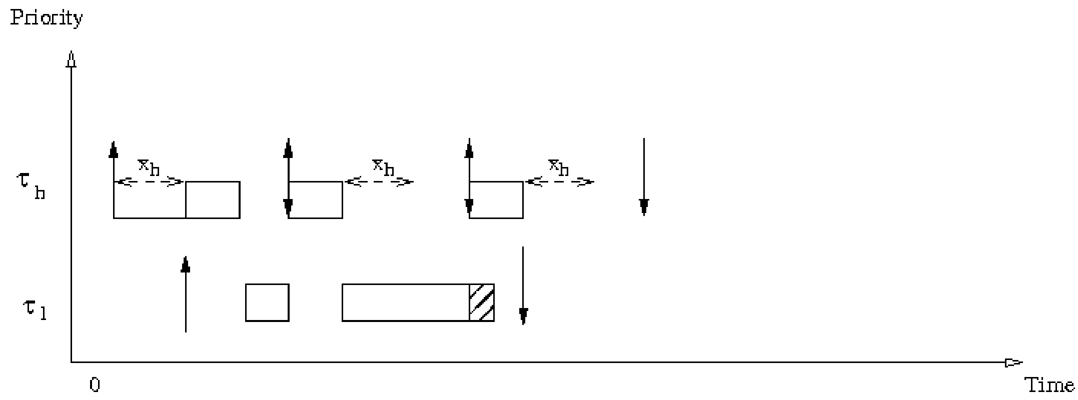

Figure 3.11.: Blocking due to self suspension. The thread τ_l misses its deadline because the first job of τ_h suspends itself. The shaded part of τ_l cannot completed before τ_l's deadline.

where bb'_i is the blocking term due to those threads for which $p_{transitive}(\tau)$ does not hold. The corresponding result for the Liu Layland criterion is:

$$\forall k \in \{1,...n\}. \sum_{i=1}^{k} \left(\frac{eb_i}{\Pi_i} + \frac{bb'_k + bb_k^{pr}}{\Pi_k} \right) \leq k \cdot (2^{\frac{1}{k}} - 1) \tag{3.11}$$

The contribution of a thread τ_h to the prohibition time is larger than its contribution to the blocking term if the scheduler would not apply *Countermeasure I* during those times when τ_h blocks or stops. In some situations, a modified admission test must therefore reject a thread set, which an admission test for an unmodified scheduler could accept. The achievable utilization of the proposed budget-enforcing fixed-priority scheduler is lower. In the worst case, the difference between the achievable utilization of an unmodified scheduler (U_{orig}) and of the proposed scheduler is as large as the prohibition time of the lowest prioritized threads:

$$U_{orig} - U = \frac{bb_{idle}^{pr}}{\Pi_{idle}} \tag{3.12}$$

Because preemptions are rare and because $max\_delay\_low(\tau)$ is small compared to the time a thread τ runs, the utilization loss due to *Countermeasure II* is negligible.

To see how the modified scheduler affects well-behaving real-time threads it is interesting to compare the different causes of blocking in admission tests with and without prohibition times. It is also interesting to relate these results to a time-partitioning scheduler.

3.5.3. Blocking due to Self Suspension

Self suspension is one reason for blocking. A thread suspends itself when it voluntarily releases the CPU to sleep for some time or when it invokes a blocking system call to wait for the arrival of a message or for the completion of asynchronous I/O.

Let x_i be an upper bound on the time the thread τ_i suspends itself. According to Liu [Liu00, Chapter 6.8.2 pg. 164ff], the blocking time due to self suspension bb_l^{ss} is

$$bb_l^{ss} = x_l + \sum_{\tau_h \in T_{high}(\tau_l)} \min(eb_h, x_h) \tag{3.13}$$

Figure 3.11 illustrates this formula in an example. A lower prioritized thread τ_l is ready whenever a higher prioritized thread τ_h suspends itself. It may therefore run any time during τ_h's second period when τ_h's second job suspends itself. Only the execution budget $eb_{h,2}$ of this job adds to the time demand of τ_l.

The blocking term of τ_l origins from the self suspension of τ_h's first job. If $\tau_{h,0}$ suspends itself for a time x_h, the critical instant of τ_l, compared to its critical instant without such a self suspension, is put off by x_h. As a result, τ_l's period overlaps the period of τ_h's third job by x_h. Hence, $eb_{h,1}$ and $eb_{h,2}$ add to the time demand of τ_l and, in addition, also the part of $eb_{h,3}$ that overlaps with the period of $\tau_{h,3}$. This part is at most as large as x_h because after x_h, τ_l's deadline ends τ_l's period.

Countermeasure I prevents lower prioritized threads from running whenever a higher prioritized thread τ_h suspends itself for which $p_{transitive}(\tau_h)$ holds. If blocking due to self suspension is the only reason for blocking, the blocking term for an admission test must be adjusted as follows.

$$bb_{cm,l}^{ss} = x_l + \sum_{\tau_h \in T_{H^-}} \min(eb_h, x_h) + \sum_{\tau_h \in T_{H^+}} \left\lceil \frac{\Pi_l}{\Pi_h} \right\rceil x_h \tag{3.14}$$

In Equation 3.14, and elsewhere, bb_{cm}^{x} stands for the blocking term bb^x, which has been adjusted to consider *Countermeasure I*. It holds that $T_{H^-} := \{\tau \in T_{high}(\tau_l) \mid \neg p_{transitive}(\tau)\}$. T_{H^+} is defined as in Proposition 1. The last addend of this equation is the prohibition time bb_l^{pr}.

Clearly, an admission test for an unmodified scheduler accepts more thread sets than a corresponding admission test for the proposed modified scheduler. The term $\left\lceil \frac{\Pi_l}{\Pi_h} \right\rceil x_h$ is typically much larger than $\min(eb_h, x_h)$. This is in particular the case if, like in RMS, the periods of lower prioritized threads are larger than the periods of higher prioritized threads.

Compared to a time-partitioning scheduler, an admission test for the proposed budget-enforcing fixed-priority scheduler accepts more thread sets because prohibition times must be considered only for threads τ for which $p_{transitive}(\tau)$ holds. A lower prioritized thread may well run during the time a higher prioritized thread suspends itself as long as the influence of this higher prioritized thread does not lead to information leakage.

A realistic scenario, in which this situation occurs, is a real-time video player for constant bit-rate videos. In this scenario, a real-time device driver reads an encrypted video that the player decrypts and displays. Assume both have the same periods and deadlines and the driver is assigned a higher priority. Then, because the driver has no plain-text access to confidential data, it is safe to classify the driver at a lower secrecy level than the player. But then, the driver runs unconstrained and the player meets its deadline as long as $eb_{driver} + eb_{player} \leq d_{player}$.

To avoid leakage from the decoder to the driver, a time-partitioning system must assign the driver to a different partition of length $eb_{driver} + x_{driver}$. But in this situation, x_{driver} additional time is required before the player's deadline d_{player} passes.

3.5.4. Blocking due to Non-preemptive Execution

The blocking time due to non-preemptive execution bb_h^{np} depends on the number of times k_h that a higher prioritized thread τ_h suspends itself. Lower prioritized threads can defer when τ_h resumes after any such self suspension by $max\_delay\_low(\tau_h)$. Moreover, lower prioritized threads can delay the resumption of τ_h after its release by this value. Hence, it holds for the blocking term bb_h^{np} of τ_h that (Liu [Liu00, Chapter 6.8.1]):

$$bb_h^{np} = (k_h + 1)\ max\_delay\_low(\tau_h) \tag{3.15}$$

On the other hand, if the proposed budget-enforcing fixed-priority scheduler applies *Countermeasure I* for τ_h, lower prioritized threads can defer only the release of τ_h. Hence, the blocking term for threads run by the proposed scheduler is

$$bb_{cm,h}^{np} = \begin{cases} max\_delay\_low(\tau_h) & \textbf{if } p_{transitive}(\tau_h) \\ (k_h + 1)\ max\_delay\_low(\tau_h) & \textbf{otherwise} \end{cases} \tag{3.16}$$

In theory, if $k_h\ max\_delay\_low(\tau_h)$ is large compared to the maximum self-suspension time x_h, admission tests for the proposed non-interference-secure scheduler could accept more thread sets than corresponding tests for unmodified schedulers. In practice however, non-preemptive critical sections are short and this effect can not be seen. Still, as far as blocking due to non-preemptive execution is concerned, admission tests for the proposed modified scheduler perform no worse than those for unmodified schedulers.

3.6. Practical Matters

In Section 3.3, I have made several assumptions, which limit the applicability of the proposed scheduler. In the following, I will partially lift these assumptions and allow threads

- to have precedence constraints (Section 3.6.1),
- to be created dynamically (Section 3.6.2),
- to hierarchically schedule other threads (Section 3.6.3),
- to donate time to other threads (Section 3.6.4), and
- to acquire resources (Section 3.7).

3.6.1. Precedence Constraints

In real-life systems, jobs typically depend on results that are produced by other jobs. As a consequence, they cannot sensibly be released before such a result is available.

Precedence constraints are one way to formally capture this dependency. Precedence constraints are typically described as a directed graph: the *precedence graph*. The vertices of this graph are the jobs. The edges denote the dependencies between jobs. That is, an edge from the job $\tau_{h,0}$ to the job $\tau_{l,0}$ denotes that $\tau_{l,0}$ depends on a result produced by $\tau_{h,0}$. In this situation, $\tau_{h,0}$ is called the predecessor of $\tau_{l,0}$. For the following discussion, I assume that the precedence graph is known at admission time and that lower prioritized threads are cleared to the precedence constraints of higher or equally prioritized threads.

There are two principle approaches to schedule threads with precedence constraints:

1. by setting the release point of a job to the point in time when the results of all predecessors are available; and
2. by setting the release point of a job to its effective release time.

The intuition behind *effective release times* is that a job is released no earlier than its predecessors. On uniprocessor systems, a scheduler, which releases jobs at effective release times, can in principle forget about precedence constraints [GJ77].

3.6.1.1. Result Dependent Release Points

The unspecified nature of release points in *ReThMo* suggests a setting of release points to the points in time when predecessor results are available. However, it is easy to see that such a setting allows predecessors to leak information by manipulating the release points of their successors: Assume $\tau_{l,0}$ is a predecessor of $\tau_{h,0}$. Then, $\tau_{l,0}$ can encode secret information in the time when it produces the result for $\tau_{h,0}$. Lower than τ_h prioritized threads τ_m can learn about this secret by observing the release of $\tau_{h,0}$.

Countermeasure I (see Definition 10 on page 61) cannot eliminate this channel if τ_h is higher prioritized than τ_l. The release of τ_h is not affected by this countermeasure.

3.6.1.2. Effective Release Times

In [Liu00, Chapter 4.5], Liu discusses two algorithms to calculate the effective release times of a thread. The basic algorithm [13] sets the effective release time of a job $\tau_{i,j}$

1. to its release point, if $\tau_{i,j}$ has no predecessors; and otherwise,
2. to the maximum effective release times of $\tau_{i,j}$'s predecessors.

Worst-case response times are not considered in this basic algorithm. In the more accurate algorithm, the effective release time of a job $\tau_{i,j}$ with predecessors is set to the maximum of the effective release times of its predecessors plus the respective worst-case response times of these predecessors.

In both versions, the release points of jobs cannot be influenced by their predecessors. However, the basic algorithm requires an adjustment of total budgets to accommodate for the blocking of predecessors. Assume τ_h is a predecessor of τ_l. Whenever τ_h blocks before having produced the desired result for τ_l, τ_l blocks also because it cannot proceed without this result. This increases the time that τ_l blocks by the time that τ_h can block. Hence, τ_l's total budget must be increased by the worst-case blocking time of τ_h.

3.6.2. Dynamic Thread Creation

In Section 3.3, I assumed that all threads are cleared to know the release points and hence the existence of higher or equally prioritized threads. However in practice, threads are often created dynamically and scheduled as aperiodic or sporadic threads.

As long as the creation of a thread is a legitimately observable event for lower or equally prioritized threads, we can reap benefit of *ReThMo*'s arbitrary but fixed release points and action

[13] As mentioned in Liu [Liu00, Chapter 4.5], the basic algorithm may have to swap jobs in the schedule to ensure their correct execution order.

traces to create threads dynamically. To do so, a dynamically created thread is inserted right from the beginning into the set of threads T. The first release point of this thread is set to the point in time of its creation. For the more general case, when threads are also created in a secret context, there are two principle approaches to schedule threads with the proposed budget-enforcing fixed-priority scheduler without revealing their existence:

1. The first is to schedule newly created threads hierarchically on top of threads that already exist in the schedule (see Section 3.6.3 below).

2. The second is to split the budgets of such an existing thread.

Assume a lower prioritized thread τ_l is not authorized to receive information from a thread τ_h. Then, if τ_h creates a thread τ_n such that τ_h and τ_n together consume at most the time that τ_h could have consumed alone, τ_l cannot distinguish whether τ_h did run or block or whether τ_n performed these actions.

A split of τ_h's total budgets fulfills the above condition if τ_n shares the priority and all release points and deadlines with τ_h. Because τ_h may encode secrets in the portion of its total budget that it transfers to τ_n, we have to require that $dom(\tau_h) \leq dom(\tau_n)$ holds for newly created threads. Also, thread creation must not allow the creator to elevate its secrecy level. Unless the creator can be trusted not to leak information in the transferred budget, we can therefore only allow the creation of equally classified threads (i.e., $dom(\tau_h) = dom(\tau_n)$ must hold).

3.6.3. Hierarchical Scheduling of Differently Classified Threads

Hierarchical CPU scheduling is an elegant way to support applications with diverse scheduling requirements in one system. The principle idea is to allow placeholder threads to act as schedulers. That is, whenever an underlying scheduler selects the placeholder, this placeholder decides which of its nested threads to run. The underlying scheduling policy and the nested scheduling policy of the placeholder can thereby differ.

In this work, I use placeholder threads merely as a vehicle to explain hierarchical schedulers. Implementations are free to implement the nested scheduling policy in a real thread that forwards its received time [FS96] or to merge the schedulers in one kernel implementation. Regehr et al. [RS01] use the latter approach for HLS. In the context of HLS, Regehr and Stankovic also discuss how the real-time guarantees of multimedia applications are preserved by hierarchical scheduling policies. Let us here focus our attention on the following two points:

- How does the proposed budget-enforcing fixed-priority scheduler preserve non-interference-properties of nested schedulers, and

- How does it avoid unauthorized information-flows between the threads it schedules and the nested threads that a placeholder thread schedules.

3.6.3.1. Avoiding Leakage due to Direct and Indirect Influence

Assume T_{nested} is the set of threads that a placeholder thread τ_p schedules. *Countermeasure I* of the budget-enforcing fixed-priority scheduler prevents higher prioritized threads τ_h from directly or indirectly influencing the placeholder thread if $dom(\tau_h) \not\leq dom(\tau_p)$. Therefore, if we set $dom(\tau_p)$ to be the greatest lower bound of the secrecy levels of the threads in T_{nested} it holds that $dom(\tau_p) \leq dom(\tau_n)$ for all $\tau_n \in T_{nested}$. Only threads τ with $dom(\tau) \leq dom(\tau_p)$ affect when the threads in the set T_{nested} are scheduled but these threads are already authorized to send to all threads in this set because $dom(\tau) \leq dom(\tau_p) \wedge dom(\tau_p) \leq dom(\tau_n) \Rightarrow dom(\tau) \leq dom(\tau_n)$.

To avoid leakage from a nested thread τ_n to threads of the underlying scheduler, we have to treat the placeholder τ_p differently. The thread τ_n can directly influence lower prioritized threads of the budget-enforcing fixed-priority scheduler only if $p_{transitive}(\tau_p)$ evaluates to false. Otherwise, whenever all threads in T_{nested} block or when they have stopped, the proposed scheduler switches to the budget consumer to avoid leakages from the nested threads that τ_p schedules. For the predicate $p_{transitive}(\tau_p)$ to evaluate to false, $dom(\tau_p) \leq dom(\tau_l)$ must hold for all threads τ_l that are lower prioritized than τ_p. Setting $dom(\tau_p)$ to the least upper bound of the secrecy levels of threads in T_{nested} authorizes direct and indirect influences only if all threads τ_n are cleared to send to these lower prioritized threads τ_l. From the least upper bound we know that $dom(\tau_n) \leq dom(\tau_p)$. Hence, $dom(\tau_n) \leq dom(\tau_l)$ holds because of the transitivity of $\leq$ and $dom(\tau_p) \leq dom(\tau_l)$.

However, $dom(\tau_p)$ is the least upper bound and the greatest lower bound of the secrecy levels of the threads in T_{nested} only if all these threads are equally classified. To support differently classified threads in T_{nested}, placeholder threads need two secrecy levels:

- $dom_{\sqcap}(\tau_p)$, to replace $dom(\tau_p)$ in $p_{transitive}(\tau_h)$ in order to determine whether *Countermeasure I* must be applied for a thread τ_h in the underlying schedule; and
- $dom_{\sqcup}(\tau_p)$, to replace $dom(\tau_p)$ in $p_{transitive}(\tau_h)$ in order to determine whether *Countermeasure I* must be applied for the placeholder thread τ_p itself.

To avoid leakage due to direct and indirect influences, $dom_{\sqcap}(\tau_p)$ is set to the greatest lower bound of the secrecy levels of the threads in T_{nested}; $dom_{\sqcup}(\tau_p)$ is set to the least upper bound of these secrecy levels.

3.6.3.2. Avoiding Leakage due to Non-Preemptive Execution

Countermeasure II avoids information leakage due to non-preemptive execution. The countermeasure predicate $p_{delay}(\tau_h)$ prevents a thread τ_l from delaying τ_h's resumption if $dom(\tau_l) \not\leq dom(\tau_h)$.

We must therefore use $dom_{\sqcup}(\tau_p)$ to determine whether *Countermeasure II* must be applied for threads τ_h in the underlying schedule. $dom_{\sqcap}(\tau_p)$ must be used to determine whether *Countermeasure II* must be applied for τ_p itself.

Because lower prioritized threads are cleared to know the release points, deadlines and total budgets of a placeholder thread, the placeholder hides the existence of dynamically created, aperiodic or sporadic threads as long as the budget-enforcing fixed-priority scheduler subjects the placeholder to both countermeasures.

3.6.4. Timeslice Donation

Hands-off scheduling [BALL90] is a technique to avoid potentially costly scheduling decisions in the performance critical IPC path. A synchronous IPC call operation (i.e., an atomic send and receive operation) blocks the caller once the callee has received the message. In situations where a higher prioritized thread calls a lower prioritized callee, the scheduler would have to check for intermediate prioritized threads. To avoid this check, hands-off scheduling allows the caller to donate the remainder of its current timeslice to the callee. Hence, the callee runs on the caller's time and priority until the next scheduling-related event occurs.

Timeslice donation in L4-Fiasco [Ste04] extends hands-off scheduling by allowing callers to provide their time until the callee replies to their request. Wolter et al. [SWH05] distinguish two forms of timeslice donation:

- *upward donation*, and
- *downward donation*.

Upward donation accounts the time the donatee runs to the donator's budget. The donatee runs on its own priority. Downward donation transfers both the time and the priority of the donator to the donatee. Hands-off scheduling is a form of downward donation, which lasts until the respective next scheduling event.

Because donatees can call other threads and because donation transfers also time when the callee is not yet ready, donators line up in a tree. The original owners of the time form the leaves of this tree, the thread that has received the donated time is located at the root of this tree. The lower part of Figure 3.14 on page 112 shows two such donation trees: e.g., in the left one τ_4 and τ_5 donate to $\tau_{R,3}$. While $\tau_{R,3}$ handles a request of τ_5, it donates to $\tau_{R,2}$ who is also receiving time from τ_2. The requests from τ_2 and from τ_4 are currently blocked (indicated by the bar at the end of the donation arrow).

Wolter et al. [SWH05] suggest to use upward donation whenever the priority of the donatee is higher than the current priority of the donator. The current priority of the donator is the maximum of the priorities received through downward donation and of the donator's own priority. Whenever the priority of the donatee is lower than the current priority of the donator, Wolter suggests to use downward donation. This way, both the stack-based ceiling resource access protocol and the priority inheritance protocol can be implemented (see Section 3.7 below).

3.6.4.1. Leakage due to Timeslice Donation

From an information-flow perspective, upward donation allows threads to leak information to so-called z-threads whereas downward donation is secure as long as the system runs the proposed budget-enforcing fixed-priority scheduler (extended with downward donation) and as long as downward donation is only between equally classified threads. Z-threads of a donator τ_s and of a donatee τ_r are those threads that run at an intermediate priority (i.e., for a z-thread τ_z it holds that $prio(\tau_s) \leq prio(\tau_z) \leq prio(\tau_r)$).

Upward Donation: Upward donation executes the donatee τ_r at its own priority. If τ_s invokes an upward donating system call to donate its time to τ_r, preemptions by z-threads τ_z are deferred until τ_r replies to τ_s.

Applying *Countermeasure I* to τ_r does not work either because then the scheduler would run τ_r or a budget consumer as long as τ_r is active. Unless τ_s is such a budget consumer, it cannot send its request to τ_r. In the discussion about precedence constraints in Section 3.6.1.1, we have already seen that messages from τ_s cannot release τ_r without introducing the possibility for information leakage.

Downward Donation: Z-threads are not affected by downward donation. The donatee runs on the time and priority of the donator. Hence, if the donator τ_s is not authorized to send to a z-thread τ_z, this z-thread cannot distinguish whether τ_s did run or whether the lower prioritized donatee τ_r consumed the time of τ_s. If τ_z is not cleared to receive information from τ_s, the countermeasure predicate $p_{transitive}(\tau_s)$ holds. The proposed budget-enforcing fixed-priority scheduler applies *Countermeasure I* to τ_s and consequently also to τ_r for as long as τ_r runs on the time of τ_s. Notice that $dom(\tau_s) \not\leq dom(\tau_z)$ precludes $dom(\tau_r) \leq dom(\tau_z)$ because otherwise, we would be able to conclude from $dom(\tau_s) \leq dom(\tau_r)$ and from the transitivity of $\leq$ that $dom(\tau_s) \leq dom(\tau_z)$ holds.

Downward donation reveals information about both, the amount of time that a donator provides to the donatee and about the amount of donated time that the donatee consumes. This constitutes a bidirectional communication channel between the donator and the donatee. However, in synchronous reliable IPC such as L4-IPC, bidirectional information flows exist anyway: the message, the message meta data and the time when the sender invokes the IPC send operation are information flows from the sender to the receiver; the time when a receiver becomes ready to receive and error situations due to too small buffers, timeouts and other error situations are information flows in the reverse direction. Therefore, the additional information flows due to downward donation are harmless because for the communication partners τ_s and τ_r to use L4-IPC, $dom(\tau_s) \leq dom(\tau_r) \wedge dom(\tau_r) \leq dom(\tau_s)$ must hold anyway.

An immediate consequence of the above observations is that we have to reject upward donation as a time-slice donation mechanism. As we shall see in the next section, this rules out the stack-based priority-ceiling resource access protocol.

3.7. Resources

In practice, threads typically require other resources besides the CPU, which they use in a mutually exclusive manner. In this thesis, I shall focus on single-unit resources. An extension of the proposed solutions to multi-unit resource accesses is left for future work.

A single-unit resource is a resource that can be held by at most one thread at a time. All other threads block on held resources. In contrast to that, threads block on a multi-unit resource only if all the equivalent copies of such a resource are already held by other threads. Critical sections are examples of single-unit resources. A device with a limited number of channels, which offers the same functionality on all channels, is a multi-unit resource.

On uniprocessor systems, short resource accesses are typically synchronized by executing the resource access non-preemptively. In Section 3.3.5, we have seen how *Countermeasure II* avoids leakage due to non-preemptive execution. Because other threads will always find that the resource is free when they are able to preempt potential resource holders, non-preemptive execution also avoids leakage due to resource contention. However, for long resource accesses, non-preemptive execution is not applicable because it negatively affects the system's response time to interrupts and other asynchronous events.

3.7.1. Self Suspension

When threads hold resources for a long time, it can happen that a higher prioritized thread preempts a resource holder. In non-real-time systems, a common strategy to react to situations in which threads cannot proceed because of held resources is to self suspend the resource acquiring thread τ_h in order to allow the resource holder to complete its operation. *Countermeasure I* jeopardizes this strategy because lower prioritized threads other than the budget consumer cannot run while the scheduler applies this countermeasure. Hence, they cannot free

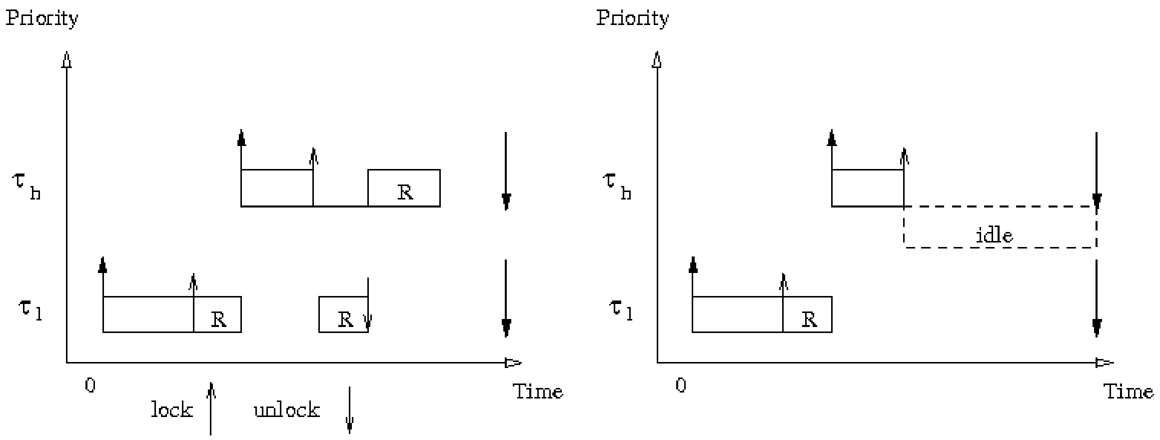

Figure 3.12.: A thread τ_h suspends itself while its required resource R is held by τ_l. Whereas self suspension of τ_h works in an unconstrained system (left), *Countermeasure I* prevents τ_l from freeing τ_h's resource (right).

the resource τ_h requires. Figure 3.12 illustrates this point.

In real-time systems, the above strategy is insufficient because the time a thread has to suspend itself is in general not bounded. For example, assume two threads τ_l and τ_l' acquire a resource R periodically that is also needed by a higher prioritized thread τ_h. Then, it can happen that the resource bounces between τ_l and τ_l' without τ_h ever getting the chance to acquire R. Real-time resource access protocols solve this problem.

3.7.2. Priority-Inheritance Protocol

The priority-inheritance protocol [SRL90] implements the following priority-inheritance rule to avoid unbounded priority inversion due to held resources. Whenever a thread τ_l blocks a higher prioritized thread τ_h because it holds a resource R that τ_h requires, τ_l inherits the time and priority of τ_h until τ_l releases R.

Sha et al. [SRL90] show that, in the absence of deadlocks, a thread has to donate at most $|R|$ time to acquire a single resource R. Here, $|R|$ is the worst-case access time of the resource R. The worst-case access time for acquiring n resources R_i in a nested fashion is $\sum_{i=1}^{n} |R_i|$. We shall return to these bounds in the discussion of information flows due to resource contention in Section 3.7.4.2. For now, let us only discuss the possible leakages of resource-acquiring threads to other threads that do not compete for resources. Because the inheritance rule of the priority-inheritance protocol resembles the priority and budget propagation of downward donation, resource accesses that are controlled by the priority-inheritance protocol cannot be used to leak information to these other threads. From an information-flow point of view, it is therefore safe to use the priority-inheritance protocol in a system with the proposed non-interference secure scheduler.

Unlike the priority-inheritance protocol, which has a bounded resource acquisition time only in the absence of deadlocks, the stack-based priority-ceiling protocol and the basic priority-ceiling protocol prevent deadlocks in the first place.

3.7.3. Stack-Based Priority Ceiling Protocol

The stack-based priority-ceiling protocol [Bak91] and the ceiling-priority protocol [Coh96] are two descriptions of the same protocol (see [Liu00, Chapter 8.6]). In the following, I shall use the formulation of the ceiling-priority protocol. The ceiling-priority protocol prevents deadlocks by running threads that hold a resource R at the ceiling priority $\hat{R}$ of this resource. The ceiling priority $\hat{R}$ is the maximum priority of all threads that require this resource.

Threads can be blocked at a resource R for at most the duration of one critical section $|R|$. However, because once a thread τ_l holds a resource it runs on the priority ceiling of this resource, τ_l can prevent intermediary prioritized z-threads from running. Hence, the stack-based priority ceiling protocol suffers from the same covert channel as upward donation. In fact, if a thread (e.g., τ_h) with $prio(\tau_h) = \hat{R}$ is used to implement the resource, upward donation to τ_h implements the stack-based priority ceiling protocol for this resource access.

3.7.4. Basic Priority Ceiling Protocol and Donation Ceiling

The basic priority-ceiling protocol [SRL90] prevents both deadlocks and covert channels to intermediary prioritized z-threads. It is defined by the following three rules (see also [Liu00, Chapter 8.5.1]):

1. *Scheduling Rule*: The current priority $\pi_\tau(t)$ of a thread τ at its release is $prio(\tau)$. τ is scheduled preemptively in a priority-driven manner according to $\pi_\tau(t)$. Rule 3 affects this priority.

2. *Allocation Rule*: Whenever τ requests a resource R that is held by another thread, it becomes blocked. Whenever τ requests a free resource R at time t one of the following two situations may occur:

   a) If the priority $\pi_\tau(t)$ is a higher priority than the current priority ceiling of the system $\hat{\Pi}(t)$, R is allocated to τ. The current priority ceiling of the system $\hat{\Pi}(t)$ is hereby the maximum of the ceiling priorities $\hat{R}$ of the resources that are held at time t.

   b) If τ's current priority $\pi_\tau(t)$ is not higher than this ceiling $\hat{\Pi}(t)$ the resource R is allocated to τ only if τ is the thread that holds a resource with a priority ceiling equal to $\hat{\Pi}(t)$.

3. *Priority Inheritance Rule*: When τ blocks on a resource that is currently held by another thread τ', τ' inherits the current priority $\pi_\tau(t)$ from τ. This inheritance lasts until τ' releases all resources with a priority ceiling equal to or grater than $\pi_\tau(t)$. At this time, the current priority of τ' drops to the value before it has acquired these resources.

Figure 3.13 illustrates the basic priority-ceiling protocol. It shows the scenario of Figure 8-10 in [Liu00, Chapter 8.5.1] extended by R_3. A thread τ_5 acquires R_2. Its current priority $\pi_{\tau_5}(t)$ remains at $prio(\tau_5)$ until the point in time t_0. At this time, τ_4 blocks because its priority is lower than the system ceiling priority $\hat{\Pi}(t) = \hat{R}_2 = prio(\tau_2)$. Because τ_4 cannot acquire R_1, τ_5 inherits the priority of τ_4 (i.e., $\pi_{\tau_5}(t) = prio(\tau_4)$). When released, the thread τ_3 runs unconstrained until it is preempted by the higher prioritized threads τ_2 and τ_1. At t_1, τ_2 blocks because τ_5 holds R_2. τ_5 inherits τ_2's priority until τ_5 frees R_2 at t_2. The thread τ_1 acquires R_1 immediately because its priority is larger than the system priority $\hat{\Pi}(t) = \hat{R}_2$. At t_2, τ_5's current priority drops to $prio(\tau_5)$ because it releases R_2. The threads τ_2 and τ_3 run to completion. At t_3, shortly after τ_2 has released R_2, the system ceiling priority $\hat{\Pi}(t)$ has dropped to the ultimately

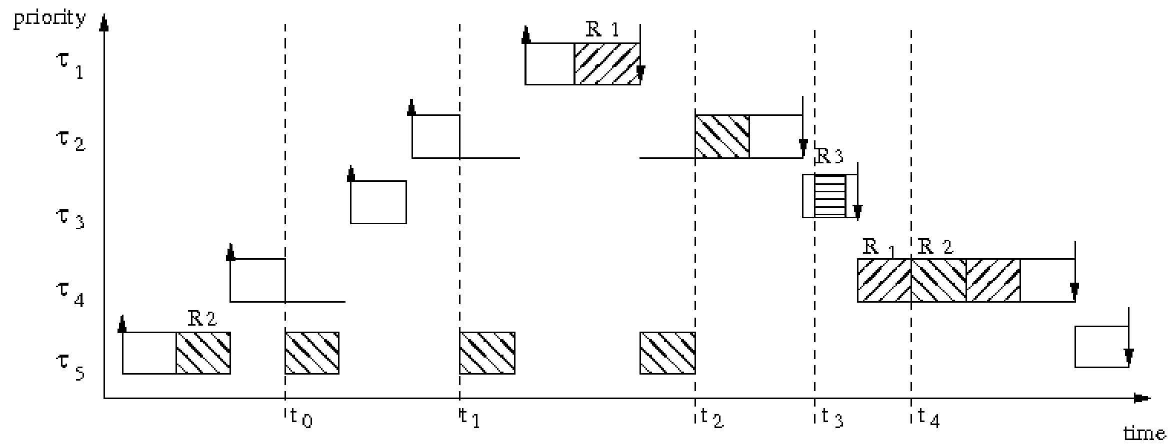

Figure 3.13.: Resource allocation according to the Basic Priority-Ceiling Protocol. The shading of shaded bars denote the resources that a thread is holding. A bottom line indicates blocking due to held resources. The dashed lines mark five points in time: $t_0, \dots t_4$.

lowest priority level of the system. The thread τ_3 may therefore acquire R_3 immediately. The thread τ_4 receives R_1 for the same reason. At t_4, the system ceiling priority $\hat{\Pi}(t_4) = \hat{R}_1$, which is higher than the current priority of τ_4. Nevertheless, τ_4 receives R_2 immediately because τ_4 holds R_1 and because the ceiling priority $\hat{R}_1$ of this resource is equal to the current ceiling priority $\hat{\Pi}(t_4)$ of the system.

In the next section, we shall see that the basic priority-ceiling protocol is secure if threads are scheduled with the proposed budget-enforcing fixed-priority scheduler. To see this point, I will introduce *donation ceiling* — an alternative description of the basic priority-ceiling protocol that is solely based on downward donation and on ceiling threads as accumulation points. The equivalence of the two protocols validates the desired property that resource acquiring threads cannot leak information to other threads if resource accesses are controlled by the basic priority-ceiling protocol.

Like the stack-based priority ceiling protocol, the basic priority-ceiling protocol and hence also the donation-ceiling protocol blocks a thread at a resource R for at most $|R|$.

3.7.4.1. Donation-Ceiling Protocol

There is a subtle difference between the ceiling-priority protocol and the basic priority-ceiling protocol. Whereas in the first, threads run immediately at the priority ceiling of a resource they acquire, the priority ceiling is used in the latter only to determine when a thread can acquire a resource. Resource holders keep their own priority until the point in time when they inherit the current priority of the threads they block. Let us for the time being assume that lack of resources is the only cause why resource holders can be blocked.

The *donation-ceiling protocol* works as follows. For each distinct ceiling priority, the donation-ceiling protocol instantiates a thread $\tau_{C,k}$, which I shall call the ceiling thread. Except that ceiling threads run only on donated time and except that they implement a significant part of the protocol, there is nothing special about these threads. Let C_k be such a ceiling priority (e.g., of two resources R_i and R_j: $C_k = \hat{R}_i = \hat{R}_j$). I say C_k is the ceiling priority of the ceiling thread $\tau_{C,k}$ if the protocol has instantiated this thread for this priority.

To request a resource R_i, a thread τ invokes the ceiling thread $\tau_{C,k}$ whose ceiling priority C_k is the smallest priority that is still higher or equal to $prio(\tau)$. All invocations in the donation-ceiling protocol are downward-donating calls. When a ceiling thread $\tau_{C,k}$ gets invoked, it executes the following protocol:

1. If the ceiling priority $\hat{R}_i$ of the requested resource R_i is higher than the ceiling priority C_k of the ceiling thread $\tau_{C,k}$, then $\tau_{C,k}$ forwards the request with a downward-donating call to the ceiling thread $\tau_{C,l}$ whose ceiling priority is the lowest priority that is higher than C_k.

2. If the ceiling priority $\hat{R}_i$ of the requested resource R_i is lower or equal to the ceiling priority C_k of $\tau_{C,k}$, then $\tau_{C,k}$ executes the resource access R_i on behalf of the requesting thread τ.

   If during this access τ requires a further resource, the ceiling thread $\tau_{C,k}$ treats the nested resource request like a new invocation. That is, if the ceiling priority $\hat{R}_j$ of the requested resource R_j is higher than the ceiling priority C_k, $\tau_{C,k}$ will forward the request to the ceiling thread $\tau_{C,l}$ with the next highest ceiling priority. If $\hat{R}_j$ is the same or a lower priority than the ceiling priority C_k, $\tau_{C,k}$ executes the request itself.

   Once $\tau_{C,k}$ releases all resources, which it has accessed on behalf of τ, $\tau_{C,k}$ replies to the call. At this time, $\tau_{C,k}$ no longer receives the time and priority from τ. After replying $\tau_{C,k}$ is ready to receive further requests.

Notice that it can never happen that a thread τ_i requests a resource from a ceiling thread $\tau_{C,k}$ that has a ceiling priority lower than $prio(\tau_i)$.

Moreover, if $\tau_{C,l}$ processes a resource request on behalf of τ_i it cannot happen that another thread τ_j obtains a resource R_k with a lower ceiling priority: If τ_i is higher prioritized than τ_j, τ_j cannot request R_k because $\tau_{C,l}$ does not block except on higher prioritized ceiling threads. If τ_i is lower prioritized, it has invoked all ceiling threads $\tau_{C,k}$ with a lower ceiling priority than C_l and a higher or the same ceiling priority than $prio(\tau_i)$. Therefore, τ_j will block when it invokes its associated ceiling thread.

The upper part of Figure 3.14 shows the same scenario as Figure 3.13. Downward donation to the resource holder is indicated by dashed arrows. The lower part of Figure 3.14 shows the donation trees for two of the four shown points in time: t_2 (left) and t_3 (right). The scenario involves 3 ceiling threads $\tau_{C,1}$, $\tau_{C,2}$, and $\tau_{C,3}$ for the ceiling priorities $\hat{R}_1$, $\hat{R}_2$, and $\hat{R}_3$, respectively.

Before t_0, no thread holds resources. All ceiling threads $\tau_{C,i}$ are therefore ready to receive resource requests. If at t_0, τ_5 requests R_2, τ_5 issues a donating call to $\tau_{C,3}$ because C_3 is the smallest ceiling priority that is still higher than $prio(\tau_5)$. When invoked, $\tau_{C,3}$ sees that $\hat{R}_2$ is a higher priority than C_3, which means it has to forward the request to $\tau_{C,2}$. Because both, $\tau_{C,3}$ and $\tau_{C,2}$, run only on donated time and priority, the current priority of $\tau_{C,2}$ is $prio(\tau_5)$. The ceiling thread $\tau_{C,2}$ sees that the ceiling priority $\hat{R}_2$ of the requested resource R_2 is equal to its ceiling priority C_2 and executes the resource access on behalf of τ_5.

When at t_1 τ_4 requests R_2, it finds $\tau_{C,3}$ waiting for the reply to the forwarded request. τ_4's donating call blocks on $\tau_{C,3}$ but the time of τ_4 is downward donated to $\tau_{C,2}$. This raises $\tau_{C,2}$'s current priority to $prio(\tau_4)$.

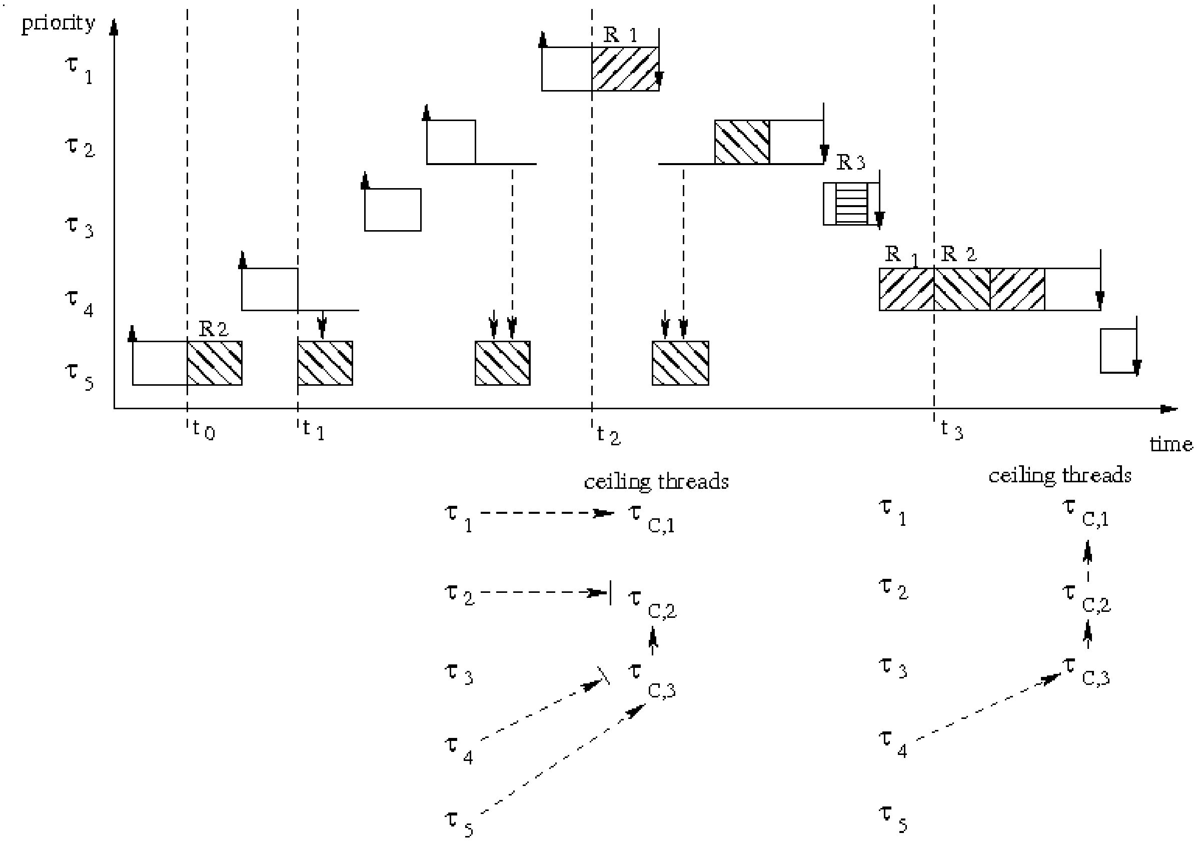

Figure 3.14.: Donation Ceiling — an alternative description of the basic priority-ceiling protocol with downward donation and ceiling threads. Dashed arrows indicate downward donations. More precisely, a dashed arrow to a shaded box indicates a downward donation of the respective thread to the ceiling thread, which handles this resource request.

At t_2, after τ_2 has also blocked on $\tau_{C,2}$, the current priority of $\tau_{C,2}$ is raised to $prio(\tau_2)$. τ_1 runs at a higher priority until it requests the resource R_1. At this time, the ceiling thread $\tau_{C,1}$ is still waiting for resource requests, which results in $\tau_{C,1}$ executing τ_1's request immediately and at the priority $prio(\tau_1)$. No lower prioritized thread runs. The lower left part of Figure 3.14 shows the donation tree at this point in time t_2. Three threads donate their time and priority to the ceiling thread $\tau_{C,2}$: τ_2, τ_4, and τ_5. τ_1 donates to τ_{C_1}. Because $\tau_{C,2}$ processes the request of τ_5, both τ_2 and τ_4 block on $\tau_{C,2}$. Nevertheless, they donate their time and priority to this ceiling thread.

At t_3, $\tau_{C,1}$ holds R_1 on behalf of τ_4. $\tau_{C,1}$ immediately continues with R_2 because the ceiling priority $\hat{R}_2$ is lower than C_1. The lower right part of Figure 3.14 shows the donation tree at this point in time. Because $\tau_{C,3}$ and $\tau_{C,2}$ have forwarded τ_4's resource request to $\tau_{C,1}$, no other thread can request a resource although $\tau_{C,1}$ runs only at τ_4's priority.

Equivalence of Donation Ceiling and Basic Priority-Ceiling To see the equivalence of the donation-ceiling protocol and of the basic priority-ceiling protocol, let us check the rules that define the latter protocol [14]:

1. *Scheduling Rule:* The basic priority-ceiling protocol runs a thread τ on its priority until a higher prioritized thread τ_h blocks on a resource held by this thread. At this time t, τ inherits the current priority $\pi_{\tau_h}(t)$ of τ_h. Donation ceiling parallels this behavior by executing the resource access in a ceiling thread $\tau_{C,k}$, which receives the time and priority of all threads that request a resource R_i with a ceiling priority $\hat{R}_i$ lower or equal to C_k.

2. *Allocation Rule:*

   a) A resource R_i is allocated to τ if τ's current priority $\pi_\tau(t)$ is higher than the system ceiling priority $\hat{\Pi}(t)$. Donation ceiling characterizes the system ceiling priority only indirectly: of all ceiling threads let $\tau_{C,j}$ be the ceiling thread with the highest ceiling priority that is executing a resource access at time t. Then, the system ceiling priority is $\hat{\Pi}(t) = C_j$. Ceiling threads with a higher ceiling priority are waiting for further requests. If $\pi_\tau(t)$ is higher than $\hat{\Pi}(t)$, τ receives R_i because its associated ceiling thread and all ceiling threads with a higher ceiling priority are awaiting τ's request. Blocking resource holders will violate this property (see below).

   b) Because ceiling threads execute resource accesses on behalf of the requesting threads, requests for nested resources are only accepted for the current resource holder at the respective ceiling priority. Moreover, because threads have to pass all ceiling threads with an intermediate ceiling priority to obtain a resource R_i, τ receives a resource in situations where $\pi_\tau(t)$ is not higher than $\hat{\Pi}(t)$ only if τ is the thread that holds a resource with priority ceiling equal to $\hat{\Pi}(t)$.

3. *Priority Inheritance Rule:* Because threads that block on a resource R_i donate in a downward-donating call their time and priority directly or indirectly to the ceiling thread responsible for R_i, donation ceiling executes resource accesses always at the highest priority of all donating threads.

Because donation ceiling parallels all rules of the basic priority-ceiling protocol precisely, both protocols are equivalent.

Implementation The above description of donation ceiling suggests a particular implementation of this protocol. Notice however that implementations are free to deviate from the protocol description as long as the behavior is preserved. For example, instead of executing resource accesses themselves, ceiling threads can return to the requesting thread in such a way that it accepts further requests only from this requesting thread. The ceiling thread can proceed with other requests only after the resource holder indicates the release of all resources for which this ceiling thread is responsible. In the current implementations of time-slice donation on L4-family kernels, the resource accessing thread must be different from the resource requesting thread.

[14]The presented equivalence proof origins from joint work with Dr. Claude-Joachim Hamann.

Blocking Resource Holders So far, we assumed that blocking on a resource is the only situation in which a resource holder blocks. To allow for other blocking situations, donation ceiling must be adjusted.

If a resource holder blocks, it can happen that a lower prioritized thread requests a resource from a ceiling thread while a ceiling thread for a higher ceiling priority is blocked. The protocol no longer avoids deadlocks.

To avoid these situations, I propose to modify donation ceiling such that ceiling threads publish when they start executing a resource access. This way, a ceiling thread can check whether all ceiling threads for a higher ceiling priority are waiting for new requests. Otherwise, it delays the request until all ceiling threads with higher ceiling priority enter this state.

3.7.4.2. Avoiding Leakage due to Resource Contention

Resource contention occurs whenever a thread attempts to acquire a resource that is held by another thread. By varying the time a resource holder occupies a resource, it can leak information to other potential resource holders.

If write authority to a resource implies read authority, it is arguable whether locking the resource for a write constitutes a covert channel at all. For read-only resource accesses, other synchronization primitives such as Reed's sequencers and event counts [RK79] can be used. Instead of blocking writers from accessing a resource, event counts allow readers to detect whether such a write has occurred concurrently to their read. In this case, they repeat the read operation until no such concurrent write has happened.

In the envisaged microkernel-based system, there are several situations where the threads of multi-level servers have to access and modify a shared resource without revealing this access to their clients. Accesses to client-spanning data structures are a prominent example of such a scenario.

The key insight that leads to a synchronization mechanism that avoids also leakage due to resource contention is that the above real-time resource-access protocols guarantee the acquisition of a resource latest after a donation of $|R|$. Hence, because the resource access itself takes at most $|R|$, a timing-leak transformation, which delays the further execution of a thread to a time $2|R|$ after the resource request, avoids leakage of this contention channel. To also prevent internal timing channels while holding R, threads must only access the resource or thread-local private memory while they hold R. This is to avoid leaking the thread-local time when the thread received R in between requesting the resource at t_r and the thread-local time $t_r + 2|R|$.

3.8. Summary

In this chapter, I have presented a budget-enforcing fixed-priority scheduler that provably avoids leakage over external timing channels even if threads have access to precise clocks. The two countermeasures that avoid this leakage are:

Countermeasure I: to treat possibly leaking threads as if they were ready in order to avoid leakage due to direct and indirect influence; and

Countermeasure II: to defer when higher prioritized threads resume their execution to avoid leakage due to non-preemptively executing lower-prioritized threads.

The resulting scheduler was formally proven non-interference secure and the proof was machine-checked with the help of the theorem prover PVS. In a first version of the scheduler, I have overlooked two corner case situations in which non-preemptively executing threads can also leak to lower prioritized threads (see Section 3.3.5.1 on page 71). The proof of the first version revealed this flaw. Based on the experience with this flaw, I expect that further adjustments of the proof to other variants of the scheduler, and in particular to the extensions proposed in Section 3.6, are straightforward. To adjust the proof for a version with timeslice donation, it is important to realize that downward donation can be seen as scheduling time quanta [15] instead of threads. The actual thread that runs on a scheduled time quanta is the thread at the root of the donation tree of this quanta, which unless trusted threads are involved must be classified at the same secrecy level as the owner of the time quanta.

The characterization of delays due to *Countermeasure I* as a blocking term allows for the reuse of a large class of existing admission tests. We have seen that the proposed scheduler preserves many real-time guarantees and that it outperforms time-partitioning schedulers — the state of the art for temporally isolated real-time systems.

The discussion of practical matters and in particular of non-interference-secure real-time resource access protocols concludes this chapter. These protocols allow multi-level servers and the microkernel to safely access shared resources without leaking secret information in internal or external timing channels that resource accesses typically imply.

In the following chapter, I introduce a static information-flow analysis for the low-level operating-system code of open microkernel-based systems. This analysis complements the proposed scheduler by establishing the absence of security policy violating information-flows in the necessarily trusted multi-level servers and in the microkernel itself.

[15] Wolter et al. [SWH05] call these objects “scheduling contexts”.

4. Statically Checking Confidentiality of Low-Level Operating-System Code

This chapter presents the second central contribution of this thesis: a sound control-flow-sensitive security type system for the low-level operating-system code of microkernel-based systems.

It is organized as follows: in the next section, I discuss the challenges of statically checking confidential-data protection in low-level operating-system code. Of these, side effects from interactions with the underlying hardware are a major obstacle. In Section 4.3, I demonstrate with the help of a simple size-aligned read why contemporary approaches to address these hardware side effects cannot scale to non-trivial amounts of operating-system code. The approach I present in this thesis consists of two steps:

- First, the to-be-checked operating-system code is translated into the non-deterministic intermediate language *Toy*.
- Then, the sound security type system for *Toy* is used to check the resulting *Toy* program and the hardware side effects for the absence of illegal information flows. The latter appear as interleaved executing *Toy* subprograms.

Section 4.5 presents the syntax and semantics of the intermediate language *Toy*. Anticipating that security type systems abstract from concrete values anyway, I have designed *Toy* to clearly separate input non-determinism from control-flow non-determinism. As a result, we only have to deal with the latter. In security-type-system-based analyses, input non-determinism comes for free.

Typically, control-flow non-determinism is addressed by standard typing rules for non-deterministic composition (see e.g., [Sab01b, pg. 45]). However, the specific nature of low-level operating-system code gives also rise to a rather unusual alternative: because only relatively small amounts of code have to be checked at a time and because this code is typically known to terminate quickly, it contains only a relatively few non-deterministic choices. Therefore, it is also feasible — though much more costly — to repeat the analysis for all possible ways in which the non-determinism in the checked program can be resolved. As we shall see in Section 4.7.1, checking all proposed alternatives leads to a much higher precision. My proposal is therefore to selectively trade precision against performance by checking all alternatives of selected non-deterministic choices and by applying the standard typing rules for non-deterministic composition to the remaining choices. In Section 4.7, I introduce the security type system for the deterministic core of *Toy*. Section 4.7.1 elaborates on typing control-flow non-determinism. To cope with shared-memory programs, I introduce in Section 4.6 the notion of learned secrets.

In Section 4.7.3, I present the soundness proof of the security type system for *Toy*. The semantics of *Toy* and the typing rules of the security type system have been formalized in the theorem prover PVS. The soundness proof of the analysis with regards to the proposed semantics has been machine checked with this theorem prover. Section 4.8 briefly summarizes this chapter.

The formal semantics of *Toy* is in part based on the kernel-code semantics of the Nova verification workpackage [TWV+08] in the European Project Robin [(Co06]. The development of this semantics was joint work with Hendrik Tews and Tjark Weber.

4.1. A Running Example

For the further discussion, let me introduce the following x86-based implementation of an artificial system call as a running example.

```
int_42_handler:
  pusha            // push all registers to stack
  call sys_add
  popa             // pop all registers from stack
  iret             // return from interrupt

void sys_add() {
  Syscall_Regs * regs = reinterpret_cast<Syscall_Regs *>(stack_top() - sizeof(Entry_Frame));

  regs->eax = regs->ebx + regs->ecx;

  sys_add_counter--;
  if (sys_add_counter == 0)
    trigger_overflow();

  asm volatile("" :: "memory");
}
```

Assuming that the kernel has properly setup the underlying hardware, the system call adds the two values in the general-purpose registers **ebx** and **ecx** and returns the result in **eax**. More precisely, when an application invokes this system call with the **int 42** instruction, the processor switches to kernel mode, pushes an entry frame on the kernel stack and invokes **int_42_handler**. As part of the entry frame, the processor pushes the user-level code segment descriptor. We shall return to this fact later in this chapter. After pushing all general-purpose registers with **pusha**, **int_42_handler** transitions control to the C++ function **sys_add**. The function **sys_add** adds the values of the pushed registers, updates the performance counter **sys_add_counter**, which keeps track of the number of invocations, and triggers an overflow exception if the preset value of this counter overflows. The compiler memory barrier **asm volatile (""::"memory");** ensures that the memory representation of all variables are up to date. At the time when **sys_add** returns, **int_42_handler** continues by popping the general-purpose registers from the kernel stack and by returning to user level with the **iret** instruction. For the following discussion, the precise layout of the two classes **Entry_Frame** and **Syscall_Regs** and the details of the involved instructions are to a large degree irrelevant. Notice however that a stack parameter defines the code segment to which **iret** returns. An incorrect setting of this parameter could result in the execution of user-level code in kernel mode.

4.2. Peculiarities of Low-Level Operating-System Code

Static information-flow analyses for low-level operating-system code have to address the following five challenges.

1. Low-level operating-system servers typically do not run in isolation. Instead, they interact with the underlying kernel, with their clients, and with other servers. Sometimes, this interaction happens in very peculiar ways. To prove the absence of information leakage, an information-flow analysis has to check all these interactions for possibilities to reveal confidential information;

2. Device drivers and large parts of the kernel interact with the underlying hardware platform. This interaction typically causes side effects through which the checked program may leak information. A sound information-flow analysis must therefore check also these side effects;

3. Operating-system programmers often use the low-level language features of C++, C, and Assembler in peculiar ways. Sometimes, they even rely on the specific behavior of a single compiler. A sound information-flow analysis and, in particular, the language semantics against which this analysis is proven sound has to consider these peculiar applications of low-level language features;

4. At the time of the analysis, the information-flow policy, to which the checked low-level operating-system code should adhere, is typically only partially known. Practical information-flow analyses must be able to deal with this impreciseness; and

5. The behavior of a system call or of a server functionality, and hence the information flows that may occur, typically depend on the parameters and on the access rights with which a client invokes these operations. However, one the one side, these parameters are typically not known at the time of the analysis. On the other side, summarizing results for all possible choices of these parameters are typically not interesting because they overestimate the possible information flows. Imprecise parameters are therefore a second source of impreciseness, a practical information-flow analysis has to deal with.

In the following sections, I investigate these challenges in greater detail.

4.2.1. Interactions with the Underlying Kernel and with other Programs

Contemporary security type systems [1] typically share the following two assumptions:

1. Interactions of the checked program with other programs are limited to the start respectively to the termination of the checked program; and

2. The code of all interacting threads is known and subjected to the same information-flow analysis.

[1] O Neill et al. [OCsC06] and Martini et al. [FM06] are exceptions.

Although the kernel and the majority of the multi-level servers never terminate while the system is running, their abstract interface specifications suggest a similar behavior for the individual invocations of these programs: Both, the kernel and the worker threads of multi-level servers are invoked by clients through a respective system call or IPC message. The parameters of both, system calls and IPC are typically passed in processor registers or in a thread-local storage area called UTCB [DLSU04]. And, the reply typically terminates the invocation and returns the result in the processor or UTCB registers. However, implementations often violate these assumptions:

- Parameters can also be passed in previously-established shared-memory regions;
- Although the kernel specification defines the UTCB to be a thread-local data structure, implementations typically map kernel-backed memory to the address space of the corresponding thread, which means other threads in this address space can also read and write this data structure. In multiprocessor systems, they may even modify the UTCB while the kernel executes non-preemptively; and
- The code of invoking clients is typically not known when a server gets analyzed.

4.2.1.1. Client-Server Interactions in L4-Based Systems

In L4-based systems, there are further, more subtle ways through which threads may interact. For instance, servers, which manage the memory of a thread, may interact with the pager of this thread by revoking this thread's memory read or write access rights. Kernel-memory managers may reclaim kernel memory to interact with the former users of objects that this reclamation destroys. And, given a thread capability, a thread can invoke the `exchange_registers` system call to trigger an exception or to cancel ongoing IPC. Through `exchange_registers`, the invoker can interact with the exception handler of the targeted thread and with the communication partners of these threads.

As these examples show, two threads may interact with each other if the capabilities held by these threads authorize system calls that read respectively modify the same kernel (or server) objects. These objects can thereby be objects to which the capabilities refer directly, or, they can be objects that are related to such a referred object. For example, for IPC gate capabilities, the related object is the thread receiving from this gate. The kernel modifies this thread as a result of delivering messages, which are send through the IPC gate. In the running example of Section 4.1, the counter **sys_add_counter** is a related object.

Whether a thread can interact with another thread depends on the system calls it is authorized to execute and on the information flows that these system calls allow. The identification of the latter is the purpose of an information-flow analysis of the microkernel. Once the information-flow policy is stated, the analysis can prove that the exercised information flows are in compliance with the security policy.

4.2.1.2. Uniform Handling of Memory Accesses and Interrupts as System Calls

To uniformly handle all interactions with the microkernel, I regard also virtual-memory accesses, interrupts and hardware exceptions as system calls. Although, admittedly, large parts of these system calls proceed without executing any kernel code.

A read or write access to some virtual address involves several access checks and a translation of this virtual address into a physical address. The necessary information for this translation is located in the processor page tables, which the kernel sets up and which the memory management unit (MMU) of the processor evaluates. If the required data is cached in the translation-lookaside buffer (TLB) or if the page-table entries convey sufficient access rights, the MMU and the load store units of the processor perform virtual-memory accesses entirely in hardware. The kernel is only involved if the page-table entries convey insufficient rights or if no valid translation is present. In this case, the MMU triggers a page-fault exception to invoke the in-kernel page-fault handler.

Actually, the MMU performs a full-fledged capability lookup when it checks the access rights in the page-table entries. The pairs (pa, R) of leaf-level page-table entries are capabilities, which refer to the memory pages at physical addresses pa and which convey R access rights.

There are two important points to notice:

1. In operating-system kernels, and in particular in an L4-based system, all user-accessible memory is effectively shared, at least with the microkernel; and hence,

2. Private memory (i.e., memory that cannot be read by other programs and that returns the stored content) is a guarantee of the kernel and of those servers that manage the memory of a thread.

The first point holds because the kernel can manipulate page tables. Hence, it can insert a mapping to any physical memory that is available in the system.

Although all memory is effectively shared, I will assume in the analysis that certain memory regions of the checked operating-system code are private and that the memory-allocation policy of the involved memory servers is free of covert channels [PN92]. The important property that reassigned memory is free of previously stored secrets can easily be established with the proposed information-flow analysis: when analyzing the kernel respectively the memory servers of our envisaged microkernel-based system, we merely have to require that all secrecy levels of the returned memory are dominated by the secrecy level of the memory requesting client.

Obviously, an all-embracing proof about the absence of illegal information flows demands also for separately-established proofs of the remaining assumptions.

4.2.2. Interactions with the Underlying Hardware

Device-register accesses, direct memory accesses (DMA), writes to special-purpose registers, modifications of hardware-traversed data structures, and the execution of privileged-mode instructions cause a variety of effects that one would not expect from executing "normal" instructions and memory accesses. Following Tews et al. [TVW09], I call these effects *hardware side effects*.

Although second-generation microkernels implement device drivers outside the kernel, drivers for interrupt controllers, timers and IO protection units have to reside inside the microkernel. As a consequence, side effects due to device-register accesses and due to DMA are also triggered by the microkernel. For example on ARM processors [Ltd], DMA is used to copy data from main memory into on-chip scratch-pad memory and back.

The following is a classification of hardware side effects by the type of behavior they cause. There are:

- side effects, which cause undefined processor behavior,
- critical side effects, and
- benign side effects.

Clearly, because undefined behavior may result in arbitrary leakages, programs that trigger side effects with such a behavior rule out any static information-flow analysis. Programs, which cause these side effects, must be rejected as potentially being insecure.

Side effects, which trigger a processor behavior that is sufficiently well defined in the processor manuals, fall into the last two classes. The determining criteria is thereby whether such a side effect rules out a further information-flow analysis. If so, the side effect is classified as a critical side effect and the side-effect triggering program must be rejected. Benign side effects may give rise to potentially harmful information flows. I call a side effect benign if a suitable static information-flow analysis can check whether these information flows violate the system's security policy.

4.2.2.1. Side Effects causing Undefined Processor Behavior

In modern processor architectures, not all instruction combinations cause a behavior that is defined in the processor manuals. For example, the Intel 64 and IA32 Architectures Software Developer's Manual specifies certain bits in the processor control registers [Cor09, § 2.5 Vol. 3a] as reserved [Cor09, § 1.3.2 Vol. 3a]. Setting these bits to a different value may cause the processor to enter an unpredictable state.

Likewise, accesses to memory-mapped device registers can cause undefined behavior. For example, accessing the 32-bit registers of the local advanced programmable interrupt controller (APIC) of an IA32 processor with loads or stores that are not 128-bit aligned or accessing these registers with floating-point instructions can cause undefined behavior. As stated in [Cor09, § 9.4.1 Vol. 3a]: "This undefined behavior could include hangs, incorrect results or unexpected exceptions, including machine checks, and may vary between implementations."

Because we do not know which information a processor leaks if it behaves in an undefined way, we have to assume pessimistically that any information is leaked. Hence, programs that cause such an undefined processor behavior have to be rejected as potentially being insecure.

4.2.2.2. Critical Side Effects

The behavior of critical side effects is well defined. However, their occurrence impedes a static information-flow analysis of programs that cause these effects.

Examples of side effects with critical state changes include page-table changes that authorize user-level programs to modify kernel code or the stack on which this code executes. A setting of bit 1 and 2 in the corresponding page-table entries to user respectively to writable [Cor09, § 3.7ff Vol. 3a] enables this side effect. Control flows to user code while in kernel mode and the disabling of paging followed by a return to user code are further side effects with critical state changes. The latter is triggered by resetting bit 31 in the IA32-CR0 register [Cor09, § 2.5 Vol. 3a] or by executing the `iret` instruction on a stack frame, which refers to a kernel-code segment.

Once arbitrary user code can be injected into the kernel, adversaries can access any information the system stores. Consequently, programs that cause critical side effects must be rejected as potentially being insecure.

Our running example in Section 4.1 modifies the variable **regs**→**eax** on the kernel stack. This stack contains also the code-segment descriptor of the invoking user-level thread. Therefore, a precondition for accepting the example as secure is that the memory locations of the entry frame and of the variable **regs**→**eax** are disjoint.

To verify the absence of critical side effects it is often helpful to observe that programs typically modify special-purpose registers and hardware-traversed data structures only during their respective initialization phase. Assuming a correct setup, we can therefore verify the absence of critical side effects by showing that no writes happen to these critical locations.

One way to perform such an analysis is to mark the corresponding fields as unmodifiable and to reject programs that write to unmodifiable fields. For our running example, the entry frame, all page tables and other hardware-traversed data structures have to be marked as unmodifiable.

When registers and hardware-traversed data structures are modified also after the initialization phase, we have to rely on separately-established correctness results. An elaborate discussion how these properties can be established with the help of static analyses is out of the scope of this thesis.

4.2.2.3. Side Effects that cause Benign State Changes

If the processor manuals describe a hardware side effect sufficiently well to allow for an information-flow analysis of this side effect, I regard it as a benign side effect.

On IA32 processors, executing a read on a virtual address v causes such a benign side effect. If this read access was the first access to the page containing v, the processor will set the accessed bits [Cor09, § 3.7.6 Vol. 3a] in those page-table entries that are involved in the translation of v to the physical address p. If we assume that p is located in RAM, the read access of v does not modify the value at p. However, the fact of reading can be leaked to other programs if these other programs are able to read the accessed bits from the used page-table entries. In the two L4-family microkernels L4-Pistachio [DLSU04] and Fiasco [Hoh02], the system call `L4-unmap` returns the accumulated accessed and dirty bits of all direct and indirect recipients of an unmapped memory page. In these systems, the setting of accessed bits constitutes an implicit information flow. The flow is implicit because only the access but not the accessed data is revealed. An immediate consequence of this information flow is that multi-level servers must not access client-provided memory in a secret context, that is, in a context with a secrecy level l_{ip} to which the respective clients τ are not cleared ($l_{ip} \not\leq dom(\tau)$). We shall return to this example in greater detail in Section 4.3 and in the case study in Section 5.1. The IA32 processor manuals describe the processor behavior on virtual memory accesses sufficiently well to allow for a formalization of the above hardware side effects and for an analysis of the information flows it involves.

Further examples of benign hardware side effects include legitimate modifications of memory-mapped device registers, the modification of the kernel stack in situations where interrupts or exceptions cause a kernel entry, and DMA transfers to memory regions that the DMA initiating driver can legitimately access [2].

[2] To enforce that DMA accesses only authorized memory regions, we must either verify this result for the part

There are three principle approaches to cope with benign hardware-side effects:

- the benign side effect and its contained information flows can be formalized and subjected to the same information-flow analysis as the side-effect triggering code;
- the program, which triggers such a benign side effect, can be rejected as potentially being insecure; or
- to checked program (respectively the kernel) can be modified to not reveal the information a benign side effect propagates.

This work advocates the first approach by formalizing hardware side effects as interleaved-executing *Toy* programs, which are checked together with the translated C++ operating-system code. However, sometimes it is also feasible to follow the last two approaches. For instance, IA32 processors support a wide range of hardware features that modern microkernels don't use. Examples include hardware task switches [Cor09, § 7.3 Vol. 3a] and real-mode kernel code [Cor09, § 17.1 Vol. 3a]. A sound analysis, which shows that the kernel does not invoke these features, relieves us from formalizing this rather complex behavior. For accessed bit propagation, the two alternatives translate into:

1. rejecting programs that set accessed bits, and
2. modifying the kernel to not return these bits.

However, both alternatives have severe drawbacks:

1. All virtual memory accesses set accessed bits, which means we would have to reject any program that accesses memory in a secret context; and
2. Without accessed-bit information, page-replacement algorithms would have to emulate these bits. However, this emulation comes at a significant performance degradation [Dra91].

To deal with programs that initiate DMA requests, I propose to treat DMA-accessible memory as "normal" shared memory. There are however three important points to notice:

1. DMA memory remains accessible even if no driver thread runs;
2. DMA does not adhere to locking schemes unless the driver holds a respective lock until the DMA transfer completes; and
3. DMA can access memory even if this memory is not accessible in the address space of the driver.

4.2.3. Low-Level Language Features in Operating-System Code

C++ [PC09] combines the high-level features of object-oriented imperative programming languages such as operator overloading, templates, inheritance, polymorphism, and exceptions with low-level data-layout and placement controls. Because the last two are inherent for operating systems [Sha06], it is quite natural that modern operating systems are typically implemented in a combination of C++, C and Assembler.

of the driver that initiates the DMA transfer [Meh05] or we have to prove the kernel to properly configure the available hardware DMA-protection units [Kru02, Int06, AJM+06].

In the development of a sound static information-flow analysis, the low-level language features of C++, and, in particular, the peculiar ways in which operating-system developers use these features, pose a challenge. The focus of this work is therefore on the low-level language features of C and C++ that appear in the low-level operating-system code of microkernel-based systems. For high-level language features, I refer the interested reader to published works about information-flow analyses of high-level language such as Java [ML98, Str03], Caml [PS03], and Haskell [LZ06].

The challenges for an information-flow analysis of low-level C++ code origin

- from the unsafe typing discipline of C++,
- from pointers and aliases,
- from non-volatile memory accesses, and
- from the non-deterministic evaluation of C++ expressions.

In the following, I shall discuss these challenges in greater detail.

4.2.3.1. Unsafe Typing Discipline

Standard semantics of programming languages (see e.g., Winskel [Win93]) are typically based on typed memory models. In such a model, variables are formalized as abstract locations, which map to the stored values. Valid pointers and references can only refer to these locations. Hence, there is no reason for a location to change its type.

C++ programs [PC09, § 1.7] and likewise C programs run on memory that is comprised of sequences of contiguous bytes. The same memory address can be accessed through different paths and possibly also with different types. Hence, the typing discipline for data types in C++ is unsafe. Consider for example the integer **int i** in **struct S** { **int i ; bool x;** } **s;**. This integer can be accessed through both, an integer pointer **int** ∗ **pi = &s.i;** or implicitly through the object **s**. In addition, the storage underlying **s** can be accessed byte wise to copy the value of **s** into another object of type **S** or to copy **s** into a character array that is large enough to hold **s** [PC09, § 3.9 pt 2,3].

In addition to these standard conform accesses, low-level operating-system code often contains accesses that are not standard conform. For instance, a typical programming pattern involves reinterpreting integer values as pointers to typed objects (e.g., to a hardware-traversed data structure). However, the involved **reinterpret_cast** conforms to the C++ standard only if the integer value is a representation of a safely-derived pointer [PC09, § 3.7.4.3 pt 3]. OS developers use this case also with other integer values. In our running example in Section 4.1, the **reinterpret_cast** to **Syscall_Regs** ∗ **regs** is an example of such a cast.

There are three important points to notice:

1. The C++ semantics and hence also the security type system for C++ operating-system code must be field sensitive. That is, types must be assigned to the individual fields of an object and not to the compound object as a whole. Otherwise, an analysis of the above example would have to regard **s** and in particular **s.x** as modified whenever values are assigned to ∗**pi**. In particular, **pi** would have to identify **s**, which is not possible without introducing auxiliary variables to hold this information;

2. The memory models must support addressing schemes that are byte granular or finer. Otherwise, copies to and from character arrays could not correctly be formalized; and

3. Addressing schemes must support all addresses that origin from reinterpreting integer values as pointers.

The assignment `*pi = h;` supports the first point. It stores a $high$ value in **s.i** but not in **s.x**. A field-insensitive information-flow analysis would classify the entire object **s** as $high$. As a consequence, such an analysis must reject programs that leak information about **s** even if they read only the boolean variable **x**.

Fortunately, the C++ standard defines all operations on classes, unions and arrays in terms of their members [PC09, § 8.5 pt 6, § 12.8 pt 8, § 12.4 pt 5]. With the exception of **vtables** [3], there is no need to maintain class objects and arrays as a whole. Instead, it suffices to keep the individual members of these objects (and the **vtable** pointer) in the memory model.

Toy inherits all data types from C++. The formalization of these data types extends the Robin data-type formalization [TWV+08] to work with a bit-granular memory model. In this formalization, a value of an interpreted data type appears as an arbitrary but fixed bitwise encoding of the object representation. The significant bits in the object representation are called the *support* of this variable. Classes and arrays are not interpreted because they are completely defined by their interpreted members.

4.2.3.2. Pointers and Aliases

In Section 2.4.7, we have already seen the benefits of a points-to analysis to determine whether two pointers refer to the same address. If so, a read through one pointer can return the value written through the respective other. In general, pointer targets may occupy the same address region, they may occupy disjoint regions or they may occupy overlapping regions. To rule out critical side effects due to modifications of hardware-traversed data structures, a particularly interesting information is whether a pointer target overlaps a region, which contains such a data structure.

In low-level operating-system code, two objects with disjoint virtual addresses can still overlap if the processor maps these addresses to the same physical addresses. Following Tews et al. [TVW09], I call these aliases *virtual-memory aliases*.

The storage of values in their bit-wise object representation already resolves most issues of aliases: If a value of type t is stored through one alias, the arbitrary but fixed encoding of this value is stored in the support bits starting at the referenced address. A subsequent read through another alias interprets bits, which are in the support of t', as an encoding of a value of type t'. Hence, if these two supports overlap, the read value depends only on the written information.

To detect the information flows through virtual-memory aliases, the control-flow-sensitive security type system for *Toy* maintains a mapping of virtual addresses to physical addresses. In this mapping, abstract physical addresses can be used as long as they correctly represent if virtual addresses are shared. For the analysis, I shall require that this mapping does not change for the checked code. That is, if the checked code accesses a variable at the virtual address v then the mapping of v to the physical address p must not change and no further mapping to p must be installed. Note, this restriction does not apply to other addresses, which the checked code does not access.

[3] Vtables are used to implement virtual functions and dynamic casts.

4.2.3.3. Non-volatile Memory Accesses

C [2105, § 6.2.5 pt 15, § 5.1.2.3 pt 5] and C++ [PC09, § 3.9.3, § 1.9 pt 9] distinguish volatile memory accesses from non-volatile accesses. The former are C++ side effects even if the volatile object is read and not modified.

The compiler is to a large degree prohibited from optimizing volatile accesses. Non-volatile memory accesses can be optimized in various ways. For example, compilers may allocate parts of non-volatile objects in processor registers, they may combine parts that are known to hold the same value, or they may omit modifications entirely if the written value is not required in the subsequent code or if this value can be derived without modifying the object.

As a result of these optimizations, the memory representation of a non-volatile object can become out-of-date. If such an out-of-date memory representation is accessed by a hardware side effect or by a concurrently executing thread, historic values can be read, which may still store a secret that has already been removed from the register-allocated part of the non-volatile object.

Admittedly, objects that are affected by hardware side effects or that are located in shared memory should be declared volatile to prevent the compiler from optimizing accesses to these objects. However in practise, programmers often avoid these declarations to allow for compiler optimizations up to the point where the data should be exchanged. At these points, compiler memory barriers such as

```
asm volatile("" :: "memory");
```

are inserted to signal to the compiler that an up-to-date memory representation is required. In combination with a proper synchronization primitive, compiler memory barriers suffice to ensure that shared-memory objects are up to date. Our running example of Section 4.1 contains such a barrier at the end of **sys_add**. It ensures that the memory representation of **Syscall_Regs** $*$ **regs** is up to date.

The challenges non-volatile objects pose on static information-flow analyses are the information flows through historic memory representations. To not risk overlooking these potential leaks, I introduce temporaries in the *Toy* intermediate language, which store intermediate results until they are explicitly written back to the processor registers, to the stack, or to the non-volatile object. The choice between these alternative locations is typically non-deterministic.

4.2.3.4. Non-deterministic Evaluation of Expressions

In [Nor99], Norrish proves that the constraints on C sequence points cause C expressions to evaluate deterministically although the standard-defined evaluation order seems to be non-deterministic [2105, § 6.5 pt 2, pt3]. The current C++ standard (though not the more recent standard drafts) adopts this definition.

However, in the presence of hardware side effects, a deterministic evaluation order of C and C++ expressions can no longer be guaranteed. Hardware side effects are not present in the C / C++ memory model on which Norrish focused in his work. It is therefore quite natural that Norrish did not consider these effects in his formal result. The following code snippet demonstrates the non-deterministic evaluation of C and C++ expressions in the presence of hardware side effects.

```
Pte * pte;
unsigned int count = 0;
...

for (unsigned int i = 0; i < 1024; i++)
  count = pte[i].accessed + count;
```

Given a pointer **pte** to the first entry of a page table, the above code snippet computes how many pages [4] have been accessed in the virtual-memory region that this page table backs. In C++, the two subexpressions **pte[i].accessed** and **count** are unsequenced [PC09, § 1.9 pt 14]. That is, they are executed in an undefined order. Assume the integer variable **count** is located in the virtual-memory region that is backed by the page-table entry **pte[0]**. Assume further that the accessed bit has been cleared prior to executing the above code snippet. Then, in the first round **i == 0**, two situations can happen:

1. If **pte[i].accessed** is executed first, the read of the integer variable **count** has not yet occurred. The expression **pte[i].accessed + count;** evaluates to zero.

2. If, on the other hand, the variable **count** is accessed first, **pte[i].accessed** reads one because, during the address translation of **count**, the MMU sets the accessed bit of **pte[0]**.

Declaring **pte[i].accessed** as volatile does not resolve this non-determinism because the hardware side effect of **count** is not part of the C / C++ memory model. For this reason, it is also quite natural that Norrish's source-level semantics cannot detect this kind of non-deterministic behavior. However, to not risk overlooking the information flows from non-deterministically evaluated expressions, this non-determinism must be supported in the semantics of *Toy* and the involved information flows must be checked with the security type system for this language.

4.2.4. Incomplete Knowledge about the Information-Flow Policy

In [HS06], Hunt and Sands introduce the universal lattice in their security type system formulation of Banerjee and Naumann's independence analysis [AB04] (see Section 2.4.4). Both, Hunt et al. and Banerjee et al., seek to prove data confidentiality of programs without precise knowledge of the information-flow policy of the systems on which these programs should run.

In this section, we shall see that Hunt's universal lattice cannot correctly characterize the information flows of shared-memory programs and for programs that access a shared kernel object. To accommodate for these programs, I will therefore extend Hunt's universal lattice with version numbers for shared-memory variables.

4.2.4.1. A Universal Lattice with Version Numbers

A universal lattice is the single lattice from which all possible typings of a program can be derived [HS06]. For programs that receive inputs only before they start, the universal lattice is the powerset of all program variables with subset as the partial order relation: $(\wp(Var), \subseteq)$. However, through shared memory and through shared kernel objects programs can also receive inputs while they are already executing.

[4] or memory regions if the page table contains also non-leaf entries

Let us consider the following artificial program p as a representative of more complex shared-memory programs:

```
1 tmp_a = shm;
2 shm = h;
3 shm = l;
4 tmp_b = shm;
5 l = tmp_a;
```

Assume the variable **shm** is located in memory that is shared with another $high$-classified program q. Assume further that q interacts only with the above program p and that it has so far only accessed low-classified information. Like before, **h** and **l** are $high$- respectively low-classified variables.

It is easy to see that the local variable **tmp_a** stores low-classified information: **shm** is only shared with q. At the time **shm** is assigned to **tmp_a**, q has seen only low-classified information. Hence, **tmp_a** must be low.

Less obvious is the dynamic secrecy level of the variable **tmp_b**. If p executes non-preemptively, q cannot access **shm** in between **shm = h** and **tmp_b = shm**. Hence, **tmp_b == shm == l**. On the other hand, if q preempts p immediately after **shm = h** (i.e., after Line 2), q can learn information about **h**. If q preempts p again after Line 3, q can return the learned information to **shm**. After **tmp_b = shm**, **tmp_b** could hold $high$-classified information. An information-flow analysis with the lattice $(\wp(Var), \subseteq)$, would however assign **l**, **tmp_a** and **tmp_b** the secrecy level $\{$**shm**$\}$. It cannot express that **tmp_a** and **tmp_b** depend on different versions of **shm**. We therefore have to extend the lattice $(\wp(Var), \subseteq)$.

Definition 16. Universal Lattice for Shared-Memory Programs
Let $LVar \subseteq Var$ be the set of local variables, $SVar = Var \setminus LVar$ the set of shared-memory variables, and let $ip \in \mathbb{N}$ range over the number of so far executed atomic steps in the following universal lattice for shared-memory programs:

$$(\wp((LVar \times \{0\}) \cup (SVar \times \mathbb{N})), \subseteq) \tag{4.1}$$

If we instantiate a control-flow-sensitive security type system with the lattice of Equation 4.1 (e.g., by replacing in Figure 2.2 on page 28 $\leq$ with $\subseteq$, $\sqcup$ with $\cup$, and by letting l and the co-domain of M and M' range over secrecy levels of the powerset $\wp((LVar \times \{0\}) \cup (SVar \times \mathbb{N}))$), we obtain a new security type system for shared-memory programs. Applied to the above program p, this security type system sets **l** and **tmp_a** to $\{$**(shm,1)**$\}$ and **tmp_b** to $\{$**(shm,7)**$\}$ because **shm** is read in the first respectively in the 7^{th} atomic step [5] in Line 1 respectively in Line 4.

As we shall see later in Section 4.6, the secrecy level of the information q may learn from **shm** — that is, the *learned secrets* of **shm** — is low until step 4 and $high$ afterwards.

4.2.5. A Protection-Parametric Information-Flow Analysis

Due to the abstractions of security type systems, it is not very helpful to check the following system call and similar programs in their entirety.

```
 if (cap->is_authorized(opcode)) {
   cap->target()->invoke(opcode);
 } else {
   return Insufficient_Access;
 }
```

[5] Assignments of the form **b = a** are assumed to take two steps: one for reading **a** and one for writing **b**.

In such a check, the information flows of all possible opcodes and the information flows of invoking the system call with both sufficient and insufficient privileges are aggregated into a single result typing environment M'. This aggregated result blurs the fact that certain information flows are legitimate only if the program has sufficient privileges. More important, if the access-control mechanism has authorized a client to invoke only some opcodes, a conforming client, which invokes only these opcodes, cannot leak information through the other opcodes. However, because the value of the parameter opcode remains abstract in the information-flow analysis, the aggregated result considers also the possible leakages of these other opcodes.

More useful are results that are parametric in the decisions of the involved access-control mechanism. To obtain these results, Banerjee and Naumann [BN03] propose to integrate a specific access-control mechanism — Java stack inspection — into a security type system. The typing rules of this type system determine whether the invoking code has sufficient privileges (on its stack) for the invoked method. However, for microkernel-based systems, this approach is not immediately applicable:

1. The invoking code, the precise security policy, and hence, the granted access rights are typically not known at the time of the analysis;

2. The allocation of kernel objects in kernel memory, the concrete distribution of capabilities, and hence, the precise addresses of targeted objects are typically not known at the time of the analysis; and

3. The capability lookups and the privilege checks are central parts of the checked system call and should therefore be subjected to the information-flow analysis.

An immediate consequence of the first point is that the authority check cannot be resolved at the time of the analysis. An immediate consequence of the second point is that we have to work with placeholder objects. To avoid blurred results, I propose to fix the access decisions and the values (or value ranges) of those parameters for which a separate information-flow result should be established. In situations where the to-be-checked code is invoked with such a parameter setting, the information-flow result can replace the more general non-parametric result. In other situations, this non-parametric result must be used or a separate result must be established for the new parameter setting. After fixing these parameters, the analysis proceeds in the following 3 steps:

1. The fixed access decisions, values, or value ranges are used as additional semantic information [TGC87] to simplify the to-be-checked program;

2. Placeholder objects are created for all parametric capability targets and for the objects that are related to these targets; and

3. The security type system is used to check the simplified program with these placeholder objects.

A sender can leak information through a system call or server function if information about an input parameter can be stored in a kernel or server object or in a related object. A receiver can learn this information if a system call or server function reveals the stored information in an output parameter or in the timing of this invocation. If placeholder objects are used for the analysis, the receiver can learn this information only if the sender accesses the same object or related object. If we know from our protection-parametric information-flow analysis that a system call leaks information into some placeholder object, an actual leakage can only occur if the access-control mechanism actually grants the sender and the receiver direct or indirect access to the same object. *Confinement* captures this last property (see Section 2.5.2 on page 39).

In the next section, I demonstrate with the help of a size aligned virtual-memory read why contemporary approaches to integrate OS functionality cannot scale to non-trivial amounts of operating-system code. As we have already seen in Section 4.2.3.4, virtual-memory reads involve a hardware side effect, which sets accessed bits during the address translation.

4.3. Typing a Size-Aligned Virtual-Memory Read

Sabelfeld [Sab01a], Mantel et al. [SM02], O Neill et al. [OCsC06] and Russo et al. [RS06] follow the same principle approach to check programs that invoke a certain functionality of the underlying operating system. Interactions with the underlying hardware are not considered, however, the same principle approach could also be applied to hardware side effects. It works as follows:

1. The semantics of the programming language is extended with a formal model of the OS functionality to consider;

2. Specialized typing rules are developed to check the information flows that this OS functionality contains; and

3. The specialized typing rules are proven sound against the extended semantics.

In the following, I demonstrate the complexity of this approach with the help of a simple, size-aligned, virtual-memory read operation.
In a microkernel-based system, programs (and typically also the kernel) run in a paged processor mode. A read of the variable a therefore involves an address translation of the variable's virtual address v_a to the physical address p_a. During this address translation, the MMU checks permission bits in the involved page-table entries and updates the accessed bits of these entries. To formalize this hardware side effect, we have to model page-table entries and their accessed bits. In addition, we have to propagate accessed bits to all virtual aliases of these page-table entries. Virtual aliases of page-table entries occur because on many processor architectures the microkernel can access page tables only if they are mapped to the kernel address space. Because the approach extends an existing type system for a programming language, we have to expect an application-level memory model. In such a memory model, the propagation of accessed bits results in complex evaluation rules for virtual-memory reads, and likewise, in complex typing rules. The evaluation rule in Equation 4.2 and the typing rule in Equation 4.3 are such rules. They are shown here merely to illustrate the complexity of contemporary approaches. The remainder of this thesis is intelligible without these rules. Figure 4.1 illustrates their basic operation.

The memory models of the standard semantics for C / C++ [Nor98, Pap98, Wal93] is a mapping from (virtual) addresses to bytes: $mem : V \to Byte$ or, more precisely, from the virtual addresses in address blocks of the form $[o, o + sizeof < T >)$ to bytes. The following is a formalization of a size-aligned virtual-memory read operation for 32-bit paging in IA32

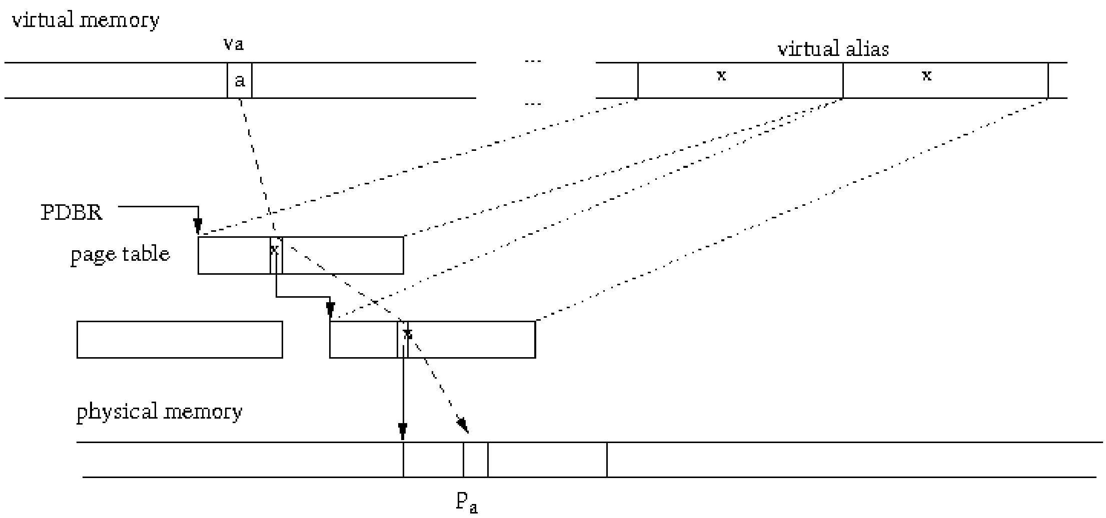

Figure 4.1.: Propagation of accessed bits to the virtual aliases of involved page-table entries.

processors [Cor09, § 4.3 Vol. 3a].

Let E_p denote the set of 4-byte aligned physical addresses at which valid page-table entries of the program p reside. The function $accessed_p : E_p \rightarrow bool$ denotes whether the accessed bits of these page-table entries are set. The partial function $pte_p : V \times Lvl \rightharpoonup E_p$ maps each virtual address $v_a \in V$ to the page-table entries used in the translation of v_a. The partial function $virt\_to\_phys_p : V \rightharpoonup P$ maintains the mapping from virtual to physical addresses. I write short $v2p$ for $virt\_to\_phys_p$. I assume the page-table base register (PDBR) is not changed while p executes. For the function pte_p, it holds that

$$(v, 2) \in domain(pte_p) \Rightarrow (v, 1) \in domain(pte_p)$$

and that

$$pte_p(v, 1) = pte_p(v', 1) \Rightarrow \lfloor \frac{v}{4MB} \rfloor = \lfloor \frac{v'}{4MB} \rfloor$$

The first property ensures that a second-level page-table entry is used for the translation of v only if there is also a first-level page-table entry. The second property ensures that the same first-level page-table entry is used to translate addresses within the same size-aligned 4MB region. However, for the following discussion, these constraints are irrelevant.

$$\frac{val = load\_var(s, v_a) \qquad s' = s\left[\begin{array}{lll} accessed(pte_p(v_a, i)) & \mapsto & true \\ mem & \mapsto & \lambda v'. \begin{cases} set\_bit(5, mem(v')) & \textbf{if } v2p(v') = pte_p(v_a, i) \\ mem(v') & \textbf{otherwise} \end{cases} \end{array}\right].}{s \xrightarrow{read(a)} (s', val)} \tag{4.2}$$

Equation 4.2 contains the evaluation rule for a memory read access. In the semantics of $read(a)$, we have to update the accessed bits of the page-table entries $pte_p(v_a, i)$, $i \in \{1, 2\}$. In addition, we have to propagate this update to all addresses v that alias the updated page-table entries. Equation 4.3 shows a corresponding typing rule for a control-flow-sensitive security type system:

$$\frac{\begin{array}{c} l_{res} = \bigsqcup_{v} M(v) \quad v_a \leq v < v_a + size \\ M' = M \left[\begin{array}{lcl} accessed(pte_p(v_a, i)) & \mapsto & l_{ip} \\ mem & \mapsto & \lambda v'. \begin{cases} mem(v') \sqcup l_{ip} & \textbf{if } v2p(v') = pte_p(v_a, i) \\ mem(v') & \textbf{otherwise} \end{cases} \end{array} \right]. \end{array}}{[l_{ip}] \vdash M \xrightarrow{read(a)} (M', l_{res})} \tag{4.3}$$

It returns the secrecy level l_{res} of the read address, updates the secrecy level of the first byte of the page-table entry, which contains the accessed bit, and it updates the secrecy levels of all addresses v' that map to this byte. The secrecy level of the byte, which contains the accessed bit, is increased by the context secrecy level l_{ip}. Because only a single bit is modified, the updates must be weak to not overlook information flows through the remaining bits of this byte. In comparison to this, the standard rule for memory reads is:

$$\frac{l_{res} = M(v(a))}{M \vdash read(a) : l_{res}} \tag{4.4}$$

The complexity of the typing rule in Equation 4.3 is immense, although the above typing rule is only for a simple size-aligned read. Reads that are not size aligned may span multiple pages. Hence, the accessed bits in two sets of page-table entries must be updated. In addition, a security typing rule for a write must propagate the written value to all virtual aliases. The rules for reads and writes to memory-mapped device registers are even more complex. A projection of this complexity to the complexity of a security type system, which is prepared to check all the low-level operating-system code of a microkernel-based system, shows the scalability limits of this approach.

A method to automatically derive sound security typing rules from a description of operating-system and hardware functionalities is therefore inevitable. Given an implementation of hardware side effects as *Toy* subprograms, the security type system for *Toy* is such a method.

4.4. Assumptions

Before I introduce the syntax and the semantics of *Toy*, let me clarify the assumptions on which the proposed information-flow analysis is based:

1. Because the individual system calls and server functionalities are checked separately, only small amounts of low-level operating-system code are checked at a time. As a result, more complex methods remain feasible for such an analysis;

2. The individual system calls and server functions terminate after a small number of atomic steps. As a consequence, all loops of the checked code terminate and the number of non-deterministic choices is bounded;

3. Due to the limited stack size, the function call depth is limited and, in particular, function calls are not recursive. When translating C++ to *Toy*, it is therefore feasible to inline all function calls;

4. As already mentioned in Section 2.4.7 on page 33, I assume a correct points-to analysis and a correct loop bound analysis;

5. The to-be-checked operating-system code must not contain null-pointer dereferences (see e.g., Tlili et al. [TD08] for a static analysis of C memory-safety properties);

6. I assume jumps in the inline assembler statements of low-level operating-system code to be trivial. By this, I mean that it is trivial to replace the jumps with appropriate if-statements; and

7. In this thesis, I focus on lock-similar programs only. That is, at any given point in time, the to-be-checked program p will hold the same locks (see Section 4.5.4.2 on page 144).

4.5. Syntax and Semantics of Toy

Toy is a simple imperative programming language with non-deterministic composition $[]$. Its syntax is given by:

Definition 17. Syntax of *Toy*

$$\begin{array}{lll}
\mathit{Types} & t & \\
\mathit{Marker} & \gamma & \\
\mathit{Temporaries} & n^t & \\
\mathit{Expressions} & e^t & := \; c^t \mid v^t \mid *(n^{addr})^t \mid n_1^t \circ n_2^t \mid \bullet\, n^t \mid ((t)\, n^{t'})^t \\
\mathit{Statements} & c & := \; e2s(n_r^t = e^t) \mid v = n^t \mid *(n^{addr}) = n^t \mid \\
 & & \; \textbf{if } (n^{bool})\ \{c_1\} \textbf{ else } \{c_2\} \mid \textbf{while}(n^{bool})\{c\} \mid \textbf{skip} \mid \\
 & & \; c_1; c_2 \mid c_1 \underset{\gamma}{\|} c_2 \mid c_1 \underset{\gamma}{[]} c_2 \mid \epsilon
\end{array}$$

A Toy *program p is a statement c. Typically, a program consists of substatements c_i and subexpressions e^t. v denotes the address of a variable, c^t abbreviates a constant of type t. $\circ$ and $\bullet$ stand for the usual unary and binary operators $+, -, *, /, <<, >>$, etc.*

I shall write **if** (n^{bool}) $\{c\}$ as syntactic sugar for **if** (n^{bool}) $\{c\}$ **else** $\{$ **skip** $\}$, $c_1 \underset{\gamma}{\diamond} c_2$ for $c_1; c_2 \underset{\gamma}{[]} c_2; c_1$ and $\textbf{skip}_{i+1}$ for $\textbf{skip}; \textbf{skip}_i$ with $\textbf{skip}_0 = \epsilon$.

Toy makes extensive use of an artificial address space of non-allocated temporaries n^t. The primary purpose of non-allocated temporaries is to hold the out-of-thin-air values of those expressions that the compiler decides to optimize away. By executing *Toy* expressions on temporaries and by non-deterministically storing these temporaries on the stack, in processor registers or not at all, the check of the proposed security type system considers a large class of compiler optimizations.

Temporaries are part of the *Toy* memory model (see Section 4.5.1 below). Without loss of generality, all *Toy* expressions except the two read expressions: v^t and $*(n^{addr})^t$, read parameters only from non-allocated temporaries. All *Toy* expressions write their computed result to the non-allocated temporary denoted by n_r^t. This temporary address is an explicit parameter of all *Toy* expressions. For a better readability, I have excluded n_r^t from all expressions in

Definition 17. With a slight abuse of notation, I shall write $n_r^t = e^t$ to denote that e^t stores its result in n_r^t. Where necessary, preceding reads and subsequent writes may read parameters from registers or from memory, or they may store results back to these locations.

Toy expressions e^t are typed. The type t is the data type of the expression result. *Toy* implements the following expressions: c^t produces the constant c of type t, v^t and $*(n^{addr})^t$ read a value of type t from a fixed address v respectively from a computed address n^{addr}. In *Toy*, $addr$ is the type for memory and register addresses. The two operators $\bullet$ and $\circ$ stand for the common unary and binary operators $+, -, *, /, <<$, etc., $((t)\ n^{t'})^t$ is a cast operator. The expression $((t)\ n^{t'})^t$ casts the t'-typed value in $n^{t'}$ into a t-typed value. As we shall see in Section 4.5.2, values of type t may be the usual values of this type (e.g., $true$ and $false$ for $t = bool$), or they may encode quiet- or signalled not-a-thing values.

Toy implements the following statements: $e2s$ allows any expression to appear as a statement. $v = n^t$ and $*(n^{addr}) = n^t$ are assignments to a fixed address v respectively to a computed address n^{addr}. **if** $(n^{bool})\ \{c\}$ and **if** $(n^{bool})\ \{c_1\}\ else\ \{c_2\}$ are the usual branching instructions, **while**$(n^{bool})\{c\}$ is the usual construct for repetition. $skip$ is a no-op statement. *Toy* implements three statements for the composition of substatements: sequential composition $c_1; c_2$, parallel composition $c_1 \underset{\gamma}{\|} c_2$, and indeterminately sequenced composition $c_1 \underset{\gamma}{\diamond} c_2$. These three statements match the three sequenced relations of the C++ standard [PC09, § 1.9 pt 14]: *sequenced before*, *unsequenced* and *indeterminately sequenced*. In addition, *Toy* implements the non-deterministic choice operator $c_1 \underset{\gamma}{[]} c_2$, which executes either c_1 or c_2. Every non-deterministic statement is marked with a marker γ. As we shall see later in Section 4.5.3.3, the marker is used to ensure that the same non-deterministic choice is always resolved in the same way. The statement ϵ denotes the empty program.

Toy lacks true while loops and function calls. The former are not required for terminating code but easily added. The only issue that remains is how to limit the number of possible non-deterministic choices and hence the number of ways in which these non-deterministic choices can be resolves (see Section 4.7.1 below). Adding function calls to the semantics and to the type system would not be difficult. An open issue though is how to adjust separately established results for non-inline functions such that they can be reused in call sites where these functions are inlined. A trivial solution is to recheck the inlined version, which ignores the performance benefit one obtains from reusing results. Other solutions to this problem are left for future work.

The formal semantics of *Toy* consists of three elements:

- a memory model,
- a data-type model, and
- a small-step semantics.

In the following sections, I describe these elements in greater detail.

4.5.1. Memory Model

Assuming that bytes are the smallest addressable units, most standard semantics for C and C++ are based on byte-granular memory models. However, many processor architectures offer instructions that read, set, clear, or complement bits individually (see e.g., [Cor09, § 3.2 Vol. 2a: BT, BTS, BTR, BTC] [6]).

In the formal semantics of *Toy*, and likewise in the formalization of the typing environment of the security type system [7], I use the following bit-granular addressing scheme:

```
Memory             : Type = [Address → Bit]
Typing_Environment : Type = [Address → Secrecy_Level]
```

To uniformly handle register accesses and memory accesses, I utilize the addressing scheme of the Robin hardware model [TWV+08]:

```
Address : Type = [# register : Register_ID, offset : nat #]
```

In this type, **Register_ID** denotes the type of the register respectively the memory. It ranges over the registers of the respective processor architecture. That is, on IA32 processors, **Register_ID** contains all general-purpose registers (such as **eax**, **ebx**, and **ecx**), all special-purpose registers (such as the page-table base register **cr3** and the code segment register **cs**), all registers of the floating-point units, and all registers of processor implemented devices (such as the registers of the Advanced Programmable Interrupt Controller (APIC)). In addition, **Register_ID** contains the identifier **Phys_Mem**. The address **(# register = Phys_Mem, offset = i #)** refers to the bit i in main memory. In *Toy*, I further extend the type **Register_ID** with two artificial register types:

- **Prem** denotes an artificial register that stores the reason for premature terminations. For example, when a loop body has exited prematurely, **Prem** contains **break** or **continue** to indicate an exit through a respective statement. **return** indicates a premature termination of a function body;

- **Temporary(id,t)** denotes the set of non-allocated temporaries that are used by the to-be-checked *Toy* program. Because a translation of a terminating C++ program to *Toy* requires only a finite amount of these temporaries, the set of non-allocated temporaries is finite for all practical purposes. Each temporary register comes with a type **t** to denote the data type of the values it may hold.

The function **valid?(r : Register_ID) : finite_set [nat]** returns the finite set of valid offsets for each register.

To avoid the complexity of propagating values to virtual aliases, the memory model of *Toy* is based on physical addresses. That is, the **offset** field of a memory address is interpreted as the physical address of the corresponding bit. Except for physically-addressed hardware data structures (such as page tables) and except for DMA transfers, whose addresses are not translated by an IOMMU, physical addresses can represent abstract locations, provided that the sharing properties of these addresses are preserved.

The microkernel and many multi-level servers leave the virtual-to-(abstract)-physical-address translation of most accessed variables constant. For these addresses, I shall abbreviate the address translation with the help of the two partial functions $v2p$ and pte that I have introduced in Section 4.3.

[6] Actually, only LOCK BTC, LOCK BTR and LOCK BTS modify single bits [Cor09, § 8.1.2.2 Vol. 3a]. The non-atomic versions of these instructions read the respective byte, modify the bit and write back the result. Concurrent memory accesses can interleave these operations.

[7] Tools are of course free to choose a more efficient representation.

4.5.2. Data Types

Data types define which operations are available on an object and how these operations interpret the memory representation of this object.

C++ comes with a rich set of predefined *fundamental types* [PC09, § 3.9] such as bool, int, and float. In addition, *compound types* such as pointers, enumerations, classes, unions and arrays allow programmers to define further types. However, because accesses to class, union or array type objects are defined member wise, there is no need to store these objects in their entirety. It suffices to store the members of these objects. Following Tews et al. [TWV+08], I call types that must be stored on their own *interpreted data types*.

Toy inherits all data types from C and C++. In addition, *Toy* defines the types `bit`, `byte`, `word` and `addr` to encode hardware side effects. In the following formalization, many aspects of data types are left abstract though.

For an interpreted data type t, the set $range(t)$ is the set of values that valid objects of this type may assume. To formalize this function, we need a supertype, which contains all possible values of objects of interpreted data types. I call this supertype *extended real*. It contains real numbers to represent floating-point types and integers to represent integral types and enums. In addition, it contains special encodings for infinity, true and false, and for the addresses of the *Toy* memory model. Hence, *extended real* is the disjoint union of real, bool, Address, and a set of special not-a-thing values (NaTs). I distinguish two types of NaTs: quiet NaTs (qNaT) and signalled NaTs (sNaT). The former encode quiet failures of operations and special results such as infinity. The latter indicate exceptions. For example, a division by zero triggers a divide-error exception (**#DE**). To be able to offload the triggering of this exception into a subsequently executed hardware side effect, the divide operator $/$ returns an sNaT value, which encodes this exception. Remember, $/$ is one of the binary operators, which I have summarized with $\circ$.

Knowing the $range$ of possible values, we can now define the semantics of interpreted data types. The following functions are left arbitrary but fixed:

Definition 18. Semantics of the Data Type t

The semantics of all data types t are characterized by $range(t)$ and by the following five functions:

$$\begin{array}{lcl} alignment(t) & : & \mathbb{N}^+ \\ size(t) & : & \mathbb{N}^+ \\ support^t & \subseteq & \{i : \mathbb{N} | i < size(t)\} \\ to\_bits^t & : & range(t) \to [support^t \to Bit] \\ from\_bits^t & : & [support^t \to Bit] \to range(t) \end{array}$$

Definition 18 extends the interpreted data type **(pod?[T])** of the Robin data-type model [TWV+08, pg. 42] to a bit-granular memory model. The functions $alignment(t)$ and $size(t)$ return the alignment respectively the size of the type t. Both values are positive (i.e., $\in \mathbb{N}^+$). The set $support^t$ denotes the set of bits that are relevant to store a value of type t. C++ allows the value representation of an object to be smaller than its object representation. That is, C++ objects can also contain holes that store no part of the object value and that are not necessarily modified. The function to_bits^t returns the bit-wise memory representation of a given value of type t. The function $from\_bits^t$ is the left inverse [8] of to_bits^t. That is, for

[8] The function $from\_bits^t$ is not necessarily the right inverse of to_bits^t. Multiple memory representations can exist for a single value (e.g., for the qNaT invalid memory representation).

$\mathit{from\_bits}^t$ and for $\mathit{to\_bits}^t$ the following condition must hold:

$$\forall\, d.\ \mathit{from\_bits}^t(\mathit{to\_bits}^t(d)) = d \tag{4.5}$$

Because the memory representation of data types and hence the above functions are implementation defined, a compiler-independent information-flow analysis would try to leave these functions abstract (of course within some reasonable bounds for the size of data types). However, unless the data types, with which memory addresses are accessed, never change, leaving the size and support of a data type abstract may result in the rejection of many secure programs. In situations where data of such a type is stored in memory, the information-flow analysis cannot precisely deduce which bits are modified. Therefore, it has to consider also information that was previously stored. The security type system would execute all writes as weak updates (see the 3rd paragraph of Section 2.4.7 on page 33).

To avoid weak updates, I propose to check *Toy* programs with concrete values for $\mathit{size}(t)$ and with a concrete support $\mathit{support}^t$ (e.g., as defined by the used compiler). The actual memory representation of values of this type (i.e., the functions $\mathit{from\_bit}^t$ and $\mathit{to\_bit}^t$) can thereby remain arbitrary but fixed. With specific encodings, the program cannot leak any additional secrets.

4.5.3. Dynamic Semantics

With the memory model and the data-type semantics in place, we can now define the small-step semantics of *Toy* programs. The semantics of *Toy* is formalized as a state transition system with states s of type State and the following three transition relations:

$\rightarrow_e$: $\mathit{Expression} \times \mathit{State} \times \mathit{Temporary} \rightarrow \mathit{State}$
for *Toy* expressions,

$\rightarrow_c$: $\mathit{Statement} \times \mathit{State} \rightarrow \mathit{Statement} \times \mathit{State}$
for *Toy* statements, and

$\rightarrow_{\mathit{ext}(io)}$: $\mathit{State} \rightarrow \mathit{State}$
to characterize the state transitions of concurrently executing threads.

The type State consists of two elements:

- the state of the memory model: $\mathit{mem} : \mathit{Address} \rightarrow \mathit{Bit}$, and
- a counter for atomic steps: $\mathit{ip} \in \mathbb{N}$.

4.5.3.1. Expressions

Figure 4.2 shows the semantics of *Toy* expressions. I use the following simplified notation for memory reads and memory updates:

- $\mathit{read}(t, \mathit{addr})$ reads the bits in $\mathit{support}^t$ starting from addr. That is, if $i \in \mathit{support}^t$, then the bit $\mathit{addr} + i$ is read from the respective register or memory. Here, $\mathit{addr} + i$ stands for the address that is obtained by increasing the offset of addr by i;

[const: c^t] $$\frac{s \rightarrow_{ext(io)} s'}{(c^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} to\_bits^t(c) \,]}$$

[read: v^t] $$\frac{s \rightarrow_{ext(io)} s'}{(v^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} read(t, v)(s') \,]}$$

[read ptr: $*(n^{addr})^t$] $$\frac{s \rightarrow_{ext(io)} s' \quad dest = from\_bits^{addr}(read(addr, n^{addr})(s'))}{(*(n^{addr})^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} read(t, dest)(s') \,]}$$

[binary op: $n_1^t \circ n_2^t$] $$\frac{\begin{array}{c} s \rightarrow_{ext(io)} s' \\ l = from\_bits^t(read(t, n_1^t)(s')) \quad r = from\_bits^t(read(t, n_2^t)(s')) \end{array}}{(n_1^t \circ n_2^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} to\_bits^t(binary\_op(\circ, t)(l, r)}$$

[unary op: $\bullet n^t$] $$\frac{\begin{array}{c} s \rightarrow_{ext(io)} s' \\ val = from\_bits^t(read(t, n^t)(s')) \end{array}}{(\bullet n^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} to\_bits^t(unary\_op(\circ, t)(val)}$$

[cast: $((t)n^{t'})^t$] $$\frac{\begin{array}{c} s \rightarrow_{ext(io)} s' \\ val = from\_bits^{t'}(read(t', n^{t'})(s')) \end{array}}{(\, ((t)n^{t'})^t, s, n_r^t) \rightarrow_e s' \,[\, mem(n_r^t) \overset{t}{\mapsto} to\_bits^t(cast(t', t)(val)) \,]}$$

Figure 4.2.: Small Step Semantics of *Toy* Expressions.

The notation $s[mem(addr) \overset{t}{\mapsto} bf]$ is used for memory updates, $read(t, addr)$ for memory reads. The relation $\rightarrow_{ext(io)}$ is the transition relation for concurrently executing threads (see Section 4.5.4.3 below). In [read: v^t], v stands for the address of a variable; $\circ \in \{+, -, *, /, etc.\}$ and $\bullet \in \{-, ++, --, etc.\}$. The functions $binary\_op$, $unary\_op$ and $cast$ characterize the behavior of the respective unary ($\bullet$), binary ($\circ$) and cast operators.

- $s \,[\, mem(addr) \overset{t}{\mapsto} bf \,]$ updates the state of the memory model mem in s at the support bits of t. These are those addresses $addr+i$ where $i \in support^t$ holds. The values of these bits are set to the values of the respective bits of the bit string bf (i.e., $s \,[\, mem(addr + i) \mapsto bf(i) \,]$ if $i \in support^t$). In situations, where not all bit addresses for the respective support bits are valid, a special qNaT value is stored in the valid bits to indicate an invalid memory access.

For better readability, I have omitted the virtual-to-physical address translation with $v2p$.

The semantics of *Toy* expressions is fairly standard: c^t returns the constant c, the expressions v^t and $*(n^{addr})^t$ perform the expected memory accesses and store the read value in n_r^t. The unary, binary, and cast expressions invoke the functions $unary\_op$, $binary\_op$ respectively $cast$, which compute the corresponding operation in the semantic domain of the respective type. For example, $binary\_op(+, int)$ performs integer addition as it is described in the C++ standard [PC09, § 5.7]. The types of these functions are:

$$\begin{array}{lcl} binary\_op(\circ, t) & : & range(t) \times range(t) \rightarrow range(t) \\ unary\_op(\bullet, t) & : & range(t) \rightarrow range(t) \\ cast(t', t) & : & range(t') \rightarrow range(t) \end{array}$$

To prove the *Toy* security type system sound against the formal semantics of *Toy* the precise computations, which $unary\_op$, $binary\_op$ and $cast$ perform, are not very interesting because the security type system abstracts from the concrete operations anyway. The security type system pessimistically assumes that any information of the parameters of these functions is encoded in their results. The security type system is therefore sound as long as semantic operations are total [9]. That is, they are defined for all possible input parameters. There are two points to notice about the *Toy* expression semantics:

1. All parameters and all results are provided in respectively stored to non-allocated temporaries. Therefore, *Toy* expressions are not recursive. More precisely, all expressions terminate in one step; and

2. In between any two atomic steps of the to-be-checked program, an arbitrary amount of steps of concurrently executing threads can execute. The relation $\rightarrow_{ext(io)}$ captures their behavior.

We shall return to the latter point in Section 4.5.4.3.

4.5.3.2. Statements

Figure 4.3 presents the small step semantics of *Toy* statements. The transition relation $\rightarrow_c$: $Statement \times State \rightarrow Statement \times State$ evaluates one atomic step of the program c on the state s. Thereby, it produces a new program c', which contains the remaining steps of c, and a new state s', which results from executing the selected step of c on s. Hence, transitions have the form $(c, s) \rightarrow_c (c', s')$. The semantics is quite standard: $e2s(n_r^t = e^t)$ executes the expression e^t, **skip** returns the result of $\rightarrow_{ext(io)}$, $v = n^t$, and $*(n^{addr}) = n^t$ update $s`mem$ as expected. $e2s$, **skip**, $v = n^t$ and $*(n^{addr}) = n^t$ complete in one step. Therefore, they all transition to the empty statement ϵ.

In the semantics of $\mathbf{if}(n^{bool})\{c_1\}\mathbf{else}\{c_2\}$ and in the semantics of $c_1 \underset{\gamma}{[]} c_2$, the execution of $\mathbf{skip}_n$ statements balances the atomic step count $s`ip$. Otherwise, the semantics of **if** is standard: dependent on the result of n^{bool}, either the if-branch c_1 or the else-branch c_2 remains as the program to be executed.

For the analysis of shared-memory programs, a balanced atomic step count is helpful to avoid unwanted correlations of memory accesses after executing if-statements with unbalanced branches. By balanced, I mean that both branches of an if-statement take the same amount of atomic steps to execute. With the definition in Section 4.5.4.3, it is easy to see that allowing concurrently executing threads to preempt a $\mathbf{skip}_n$ statement multiple times is equivalent to allowing these threads to preempt $\mathbf{skip}_1 = \mathbf{skip}$. Essentially, $\rightarrow_{ext(io)}$ allows concurrently executing threads to execute non-deterministically for a not further specified amount of time. Therefore, provided that the to-be-checked program does not do anything, we can always summarize the executions of multiple preemptions into the executions of one single preemption.

The *Toy* semantics of while-loops is defined by the unfolding rule: execution of a while-loop is equivalent to executing an if-statement, which checks the condition of the while-loop and executes the body plus the while-loop if the condition evaluates to true and which finishes the loop otherwise. However, in our specific setting, only terminating system calls or server invocations are analyzed. Therefore, we can reap-benefit of correct loop-bound analyses (see Section 2.4.8) to obtain a much simpler semantics for terminating while-loops.

[9] Pottier and Simonet [PS03] apply the same trick to simplify their semantics of ML.

[expr: $e2s(n_r^t = e^t)$] $$\frac{(e^t, n_r^t, s) \rightarrow_e s'}{(e2s(n_r^t = e^t), s) \rightarrow_c (\epsilon, s')}$$

[skip] $$\frac{s \rightarrow_{ext(io)} s'}{(\textbf{skip}, s) \rightarrow_c (\epsilon, s')}$$

[write: $v = n^t$] $$\frac{s \rightarrow_{ext(io)} s'}{(v = n^t, s) \rightarrow_c (\epsilon, s' \, [\, mem(v) \overset{t}{\mapsto} read(t, n^t)(s') \,])}$$

[write ptr:] $$\frac{s \rightarrow_{ext(io)} s' \quad dest = from_bits^{addr}(read(addr, n^{addr})(s'))}{(*(n^{addr}) = n^t, s) \rightarrow_c (\epsilon, s' \, [\, mem(dest) \overset{t}{\mapsto} read(t, n^t)(s') \,])}$$

[if (true)] $$\frac{s \rightarrow_{ext(io)} s' \quad from\_bits^{bool}(read(bool, n^{bool})(s')) = true}{(\textbf{if } (n^{bool}) \; \{c_1\} \textbf{ else } \{c_2\}, s) \rightarrow_c (c_1; \textbf{skip}_{\max(|c_2|-|c_1|,0)}, s')}$$

[if (false)] $$\frac{s \rightarrow_{ext(io)} s' \quad from\_bits^{bool}(read(bool, n^{bool})(s')) = false}{(\textbf{if } (n^{bool}) \; \{c_1\} \textbf{ else } \{c_2\}, s) \rightarrow_c (c_2; \textbf{skip}_{\max(|c_1|-|c_2|,0)}, s')}$$

[while:] $$\frac{}{(\textbf{while}(n^{bool})\{c\}, s) \rightarrow_c (\textbf{if}(n^{bool})\{c; \textbf{while}(n^{bool})\{c\}\}, s)}$$

[seq comp: $c_1; c_2$] $$\frac{(c_1, s) \rightarrow_c (c_1', s')}{(c_1; c_2, s) \rightarrow_c (c_1'; c_2, s')}$$

[epsilon comp: $\epsilon \times c_2$] $$\frac{\times \in \{\, ; \; \| \,\}}{(\epsilon \times c_2, s) \rightarrow_c (c_2, s)}$$

[choice left: $c_1[]c_2$] $$\frac{pick(\gamma) = true}{(c_1 \underset{\gamma}{[]} c_2, s) \rightarrow_c (c_1; \textbf{skip}_{\max(|c_2|-|c_1|,0)}, s)}$$

[choice right: $c_1[]c_2$] $$\frac{pick(\gamma) = false}{(c_1 \underset{\gamma}{[]} c_2, s) \rightarrow_c (c_2; \textbf{skip}_{\max(|c_1|-|c_2|,0)}, s)}$$

[parallel left: $c_1 \parallel c_2$] $$\frac{pick(\gamma_i) = true \quad (c_1, s) \rightarrow_c (c_1', s')}{(c_1 \underset{\gamma_i}{\parallel} c_2, s) \rightarrow_c (c_1' \underset{\gamma_{i+1}}{\parallel} c_2, s'')}$$

[parallel right: $c_1 \parallel c_2$] $$\frac{pick(\gamma_i) = false \quad (c_2, s) \rightarrow_c (c_2', s')}{(c_1 \underset{\gamma_i}{\parallel} c_2, s) \rightarrow_c (c_1 \underset{\gamma_{i+1}}{\parallel} c_2', s')}$$

Figure 4.3.: Small Step Semantics of *Toy* Statements

The notation for memory reads and updates and for $\rightarrow_{ext(io)}$ are as described for Figure 4.2. The oracle $pick$ is used to resolve the control-flow non-determinism of the *Toy* statements $[]$ and $\|$ (see Section 4.5.3.3).

Let $i = loop\_bound(s`ip)$ be the upper bound on the number of iterations of the while loop $(\mathbf{while}(n^{bool})\{c\}, s)$ as returned by the used loop-bound analysis. Then, $\mathbf{while}(n^{bool})\{c\}$ can be written as $\mathbf{while}^i(n^{bool})\{c\}$ with the following evaluation rules:

$$[\text{while: } \mathbf{while}^0] \qquad \frac{}{(\mathbf{while}^0(n^{bool})\{c\}, s) \rightarrow_c (\mathbf{skip}, s)}$$

$$[\text{while: } \mathbf{while}^i] \qquad \frac{i > 0}{(\mathbf{while}^i(n^{bool})\{c\}, s) \rightarrow_c (\mathbf{if}(n^{bool})\{c; \mathbf{while}^{i-1}(n^{bool})\{c\}\}, s)}$$

The consequences for the *Toy* security type system are the following. After applying the rule [while: $\mathbf{while}^i$] i times, the to-be-checked *Toy* program contains only if-statements (and $\mathbf{while}^0 = \mathbf{skip}$) instead of $\mathbf{while}(n^{bool})\{c\}$. Hence, the more precise typing rule for if-statements can be used to check the program for illegal information flows. Although their addition would be sound, a security type system for low-level operating-system code does not necessarily require typing rules for true while loops such as Rule C4 and Rule C4' in Figure 2.2 on page 28.

The semantics of the sequence statement $c_1; c_2$ is standard. It evaluates first c_1 until no more atomic steps are left on the left side of ; (i.e., until $\epsilon; c_2$ remains to be executed). At this time, the rule [epsilon comp] collapses $\epsilon; c_2$ to c_2.

4.5.3.3. Control-Flow Non-determinism in *Toy*

The intuition behind control-flow non-determinism is that it represents under-specification, which can be resolved by the implementor or by some mechanism at run-time. For example, we have seen that the evaluation order of most C++ expressions is non-deterministic. The compiler resolves this non-determinism by putting the corresponding assembler statements into the binary in a particular order. The interleaving of hardware side effects with the program control flow is typically resolved by a run-time mechanism. For example, the interleaving of reads and writes of the to-be-checked program with DMA memory accesses is resolved at runtime by the arbitration logic of the memory bus and by the processor caches.

Toy implements two non-deterministic statements: $[]$ and $\|$. The statement $c_1 \underset{\gamma}{[]} c_2$ either resolves to c_1 or to c_2, the choice of which is non-deterministic. As a consequence, indeterminately sequenced composition $c_1 \underset{\gamma}{\diamond} c_2$ either executes c_1 before c_2 or c_2 before c_1 (i.e., $c_1 \underset{\gamma}{\diamond} c_2 = c_1; c_2 \underset{\gamma}{[]} c_2; c_1$). Parallel composition $c_1 \underset{\gamma_i}{\|} c_2$ produces an arbitrary interleaving of c_1 and c_2. For that, either c_1 is chosen non-deterministically to execute one step or c_2 is chosen to advance by one step. Like if-statements, $c_1 \underset{\gamma}{[]} c_2$ balances the atomic step count $s`ip$ by executing $skip$ statements after c_1 respectively after c_2.

To formalize these non-deterministic binary choices, I use an arbitrary but fixed function $pick$, which acts as an oracle and which either returns $true$ or $false$. In the definition of $pick$, it is tempting to choose $State$ as the domain of this function [10]. However then, refinements of $pick(s)$ can depend their decisions on arbitrary secrets in s.

[10] The monotonic atomic step count $s`ip$, which is part of the state s, ensures that a *Toy* program never returns to the same state.

Consider for example the following program p:

```
int a;
int l;

(a = h) || (l = 0 [] l = 1)
```

Intuitively, one would expect this program to be secure because only constants are assigned to the *low*-observable variable **l**. However, if an implementor decides to resolve the non-deterministic choice such that it favors **l = 0** if, lets say, **a** $<$ **5** and in favor of **l = 1** otherwise, information about the value in **h** can be leaked.

To avoid these complications, Lowe [Low04] introduces a marker scheme, which ensures that non-deterministic choices are always resolved in the same way. Based on this marker scheme, $[]$ is marked as a non-deterministic choice that evaluates independently from the non-deterministic choice that determines the interleaving with **a = h**. In this work, I adopt the same principle idea by allowing $pick$ to depend only on markers of the form γ, respectively on markers of the form γ_i for parallel composition. Hence, $pick$ is a function of type $Marker \rightarrow bool$.

In the transition rule of $\|$, the non-determinism cannot be resolved immediately as it is the case for $[]$ and hence also for $\diamond$. As a consequence, we cannot apply the same simple marker scheme because transition rules of the form:

$$[\text{parallel left: } c_1 \parallel c_2] \qquad \frac{pick(\gamma) = true \quad (c_1, s) \rightarrow_c (c_1', s')}{(c_1 \underset{\gamma}{\parallel} c_2, s) \rightarrow_c (c_1' \underset{\gamma}{\parallel} c_2, s'')}$$

$$[\text{parallel right: } c_1 \parallel c_2] \qquad \frac{pick(\gamma) = false \quad (c_2, s) \rightarrow_c (c_2', s')}{(c_1 \underset{\gamma}{\parallel} c_2, s) \rightarrow_c (c_1 \underset{\gamma}{\parallel} c_2', s')}$$

would always pick atomic steps of one of the statement c_1 respectively of c_2. To avoid these complications, I use markers of the form γ_i, which keep track of the so far executed atomic steps in $c_1 \parallel c_2$. Initially, *Toy* programs contain only parallel-composition statements of the form $c_1 \underset{\gamma_0}{\parallel} c_2$.

For the definition of $\rightarrow_{ext(io)}$, which completes the formal semantics of *Toy*, we have to characterize the interactions between concurrently executing threads and the to-be-checked program. These interactions can be through shared memory or through other shared kernel objects.

4.5.4. Shared Memory

In the information-flow analysis of the microkernel and of its multi-level servers, it is our primary concern to establish that:

1. the checked program p does not leak internal secrets; and

2. the checked program p does not forward secrets from one client to another client unless the latter is cleared for this information.

To enforce the second point, we have to characterize the possible information flows between clients of the checked program and between the checked program and these clients. Let us first focus on information flows through shared memory.

Clients with read authority to some memory page, which is mapped to the checked program's address space, can learn any information about the data that p stores in this page. This is of course provided the location is not locked or provided the client accesses the memory location without first acquiring the lock. In the latter case, I say the client does not adhere to the locking discipline for this memory location.

When scheduled, a client may forward any information it can read over any channel that is available to it. Moreover, it may store the read information to forward it at a later point in time. Similar to reading shared memory, a client with write access to a shared page can write previously learned information in this page. The encoding of this information can thereby be arbitrary. Apart from that, if a client is authorized to access a page but if it did not yet exercise this authority to obtain a respective capability, it may first request such a capability and then access the referred page. In this case, I say the client has *potential access* to the page.

4.5.4.1. A Formal Model of Shared Memory Interactions

To formalize the interactions through shared memory, we have to characterize the propagation of information in the addresses of p that are read shared with other threads. This information can propagate over a chain of communicating threads into write-shared addresses. The threads in this chain may remember previously read information. The relation $\mathit{effects}_p$ formalizes this propagation of information:

Definition 19. Effects

Let p be the checked program, T the set of threads that directly or indirectly interact with p. R^p_τ is the subset of addresses A of p to which $\tau \in T$ has potential read access. $W^p_\tau \subseteq A$ denotes the set of potentially write-shared addresses with τ. The relation τ can_send^* τ' holds if τ can directly or indirectly send to τ'. Then, concurrently executing threads can forward information in the read-shared address a to the write-shared address a' if a and a' are related by:

$$a \ \mathit{effects}_p \ a' \ :\Leftrightarrow \ \exists\, \tau, \tau' \in T.\, a \in R^p_\tau \ \wedge \ a' \in W^p_{\tau'} \ \wedge \ \tau \ can\_send^* \ \tau'$$

The relation can_send^* is the transitive, reflexive closure of can_send. τ can_send τ' denotes that τ can directly send information to τ' without the help of a third thread. For the time being, this is the case if τ has write access to some memory to which τ' has read access.

Obviously, an information-flow analysis of p is only sensible if p is checked against an environment that does not already allow security policy violating information flows. Otherwise, the analysis would reject p as potentially insecure even if p is not involved in the leakage of any information. Definition 20 formalizes this requirement:

Definition 20. Proper System Configuration

The set of concurrently executing threads T is properly configured if it fulfills the following condition:

$$\forall\, \tau, \tau' \in T. \tau \ can\_send^* \ \tau' \ \Rightarrow \ dom(\tau) \leq dom(\tau')$$

Like before, $dom(\tau)$ is the classification of the thread τ.

4.5.4.2. Locking Shared-Memory Addresses

Provided that concurrently executing threads adhere to the locking discipline, locks can temporarily protect the data stored in shared-memory addresses. However, at the same time, many lock implementations reveal when the checked program p holds such a lock. Moreover, shared memory can reveal when p holds a lock even if the lock itself hides this fact.

To avoid information leakage due to lock acquisition, a common approach is to classify locks and to reject programs that acquire locks in secret contexts. For example, Sabelfeld [Sab01a] rejects programs that acquire locks in any context except the lowest classified context $\perp$. Russo et al. [RS09] tolerate acquisitions in higher classified contexts only if the scheduler is signalled to prevent the interleaved execution of lower classified threads together with higher classified threads. In our setting, there is a further alternative. In Section 3.7, we have seen two lock implementations that, when used in combination with the proposed information-flow secure scheduler, do not reveal whether a lock is held by another thread. These locks can be taken by any thread and in any context. Let us therefore focus on locks of this latter class.

Even though the lock itself may not reveal when it is taken, shared-memory accesses may leak this information. Consider for example the following program p with two shared-memory variables **shm_a** and **shm_b** and a lock **l** that protects **shm_a**:

```
1  shm_a = 0; shm_b = 0;
2
3  if (h) {
4    lock(l);
5      shm_b = 1;
6      shm_b = 2;
7    unlock(l);
8  } else {
9      shm_b = 1;
10     shm_b = 2;
11 }
12
13 l = shm_a;
```

Intuitively, one would accept this program as secure unless a concurrently executing thread τ with potential write access to **shm_a** has access to $high$-classified information. Even if **shm_b** is only read shared with τ, one would consider p as secure because p executes **shm_b = 1; shm_b = 2;** independent of the value of **h**. However, if the concurrently executing thread τ executes the program q:

```
old = shm_b;

while(old != shm_b) {
  lock(l);
    shm_a++;
  unlock(l);
  old = shm_b;
}
```

the value of **h** may be leaked to **l**. If **h == false** and τ preempts the execution of p before Line 8, immediately after Line 9 and again after Line 10, τ detects two modifications of **shm_b** and updates **shm_a** twice. If, on the other hand, **h == true** and τ preempts the execution of p before Line 4 and immediately after Line 5, it detects the change from **shm_b == 0** to **shm_b == 1**, however before it can execute **old = shm_b** it has to wait for p to release the lock on **shm_a**. Similarly, if τ preempts the execution of p immediately after Line 6, it only detects the change from **shm_b == 0** to **shm_b == 2**. Hence, it increases **shm_a** only once. If p would also lock **shm_a** while executing **shm_b = 1; shm_b = 2;** in the else branch, a low-classified observer is no longer able to distinguish the two branches.

In this thesis, I will therefore focus on *lock-similar* programs.

Definition 21. lock-similar programs

A program p is lock similar *if in any two states s and t with identical atomic-step count $s`ip$, $t`ip$ holds the same locks.*

Let L_p be the set of locks that p may acquire. The function

$$locks_p : \mathbb{N} \rightarrow \wp(L_p) \tag{4.6}$$

denotes for each atomic step $i \in \mathbb{N}$ the locks held by p. The function

$$locked\_addresses_p : L_p \rightarrow \wp(A) \tag{4.7}$$

defines for each lock l the subset of the addresses in A, which this lock protects. Only those addresses a are considered as locked addresses that are shared with threads that adhere to the locking discipline. That is, such an adhering thread τ guarantees not to access a while the protecting lock l is held by p.

Certain locks enforce an adherence to the locking discipline even if unchecked and thus potentially untrustworthy programs attempt to access shared data. For example, the disabling of preemptions on a uniprocessor systems protects all memory addresses that are not DMA accessible. A server, which grants access to lock-protected memory only to the current lock holder and which guarantees not to access locked pages, implements a similar protection.

Alternatively, programs can be statically checked for correct lock usage. For example, in [IK08], Iwama and Kobayashi present a type system, which performs such a check for Java bytecode.

Because lock-protected addresses can only be accessed by p, they are temporarily only accessible by p. The function $local_p$ combines $locks_p$ and $locked\_addresses_p$ to denote these addresses.

Definition 22. Local Addresses

The function $local_p$ defines for each step, which addresses are protected by a lock that p holds. It is defined as:

$$\begin{aligned} local_p &: \quad \mathbb{N} \rightarrow \wp(A) \\ local_p(i) &:= \bigcup_{l \in locks_p(i)} locked\_addresses_p(l) \end{aligned}$$

The functions $locks_p$ and $locked\_addresses_p$ are given by Equation 4.6 and Equation 4.7, respectively. For an address to be in $locked\_addresses_p$, it must be shared only with those threads that adhere to the locking discipline.

4.5.4.3. Input Non-Determinism and Concurrently Executing Threads

Unless the to-be-checked program p holds a lock for an address a, all concurrently executing threads $\tau \in T$, which share a in a writable fashion, can write an arbitrary value to a. In particular, they may store in a an arbitrary encoding of information they have learned from the previous execution of p or from other threads. They may however also decide to leave a unchanged.

To formalize these inputs let me introduce a second oracle: the input oracle io. The function io returns for each write-shared address $a \in \bigcup_{\tau \in T} W^p_\tau$ either a value of type Bit or the special symbol nil to denote that a is not modified. I assume that nil is not contained in the set $\{0, 1\}$.

Hence, the type of the co-domain of io is $Address \to \{0,1\} \cup nil$. Obviously, several threads may modify an address a in between two atomic steps of the to-be-checked program p and each such thread may modify a several times. Therefore, if io returns a value of type Bit for a, I assume that this value reflects the last modification of a prior to p resuming its execution.

The challenge in the definition of io is to allow for both, arbitrary encodings and l-similar inputs, if the information-flow analysis checks p for information leakage to l-classified observers. We shall return to this last point in Section 4.6.3. For now, let io be an arbitrary but fixed function with the three parameters l, L^i and s, where s is a state, $i = s\text{`}ip$, l is the observer secrecy level, and L^i is the secrecy level of the learned secrets that can be returned to a. In Section 4.6, I shall introduce learned secrets in greater detail.

The rule for updates of write-shared memory addresses by concurrently executing threads follows from $local_p$ and from the input oracle io:

Definition 23. Shared-Memory Updates by Concurrently Executing Threads
Let T be the set of concurrently executing threads, W_τ^p the set of write-shared addresses with such a thread $\tau \in T$, and let io be an input oracle. Then, the transition rule for concurrently executing threads $\to_{ext(io)}$: $State \to State$ is defined as follows:

$$s \to_{ext(io)} s' \quad where$$

$$s' \;=\; s \left[\begin{array}{lll} ip & \mapsto & ip(s)+1 \\ mem & \mapsto & \lambda a. \begin{cases} io(l,L,s)(a) & \textbf{if} \begin{array}{l} io(l,L,s)(a) \neq nil \wedge \\ a \notin local_p(s\text{`}ip) \wedge \\ a \in \bigcup_{\tau \in T} W_\tau^p \wedge \\ register(a) = Phys\_Mem \end{array} \\ s'mem(a) & \textbf{otherwise} \end{cases} \end{array}\right]$$

This completes the definition of the *Toy* semantics except that we have to notice that threads can exchange information also via other shared kernel or server objects.

4.5.4.4. Other Kernel Objects

In the relation can_send and hence in $effects_p$, we have so far only considered information flows through shared memory. In this section, I extend can_send to also consider information flows through shared kernel or server objects.

In situations where a concurrently executing thread τ causes the modification of a variable of a kernel or server object, τ may leak confidential information into this object. Now, if another concurrently executing thread τ' obtains read access to the same variable of the same object, or if it can otherwise learn about the information in this variable, τ can send information to τ'. Thereby, the shared variable can reside in the objects that are targeted directly by the capabilities with which τ respectively τ' invoke their system calls or server invocations. Or, the variable can reside in an object that is related to the targeted object. To detect these information flows, the respective system calls (or server invocations) can be checked with the security type system for *Toy*.

If these calls are checked with the universal lattice for shared-memory programs (Definition 16), τ can send to τ' if the following two conditions hold [11]:

1. If the identifier or a variable of a placeholder object appears in a secrecy level of the output parameters of the call τ' invokes, τ' can read this parameter. Remember that, in a check with the universal lattice, a secrecy level is the set of variable identifiers that contribute to the stored result respectively the set of pairs of variable identifiers and step counts if the identifier denotes a shared-memory variable.

2. The thread τ may write information to one of these variables if the secrecy level of this variable contains an input parameter of the system call or of the server function τ invokes.

If these two conditions hold, I extend the relation can_send by the pair (τ, τ'). That is, in the extended relation it holds that $\tau\ can\_send\ \tau'$. To give an example of such a leakage, let us consider IPC between τ and τ'. If τ invokes the IPC-send operation on a suitable communication channel on which τ' is receiving, the kernel modifies a kernel object that is related to this communication channel: the message registers in the UTCB of the receiver τ'.

In situations where the checked program p invokes a system call or server function to read or write a variable in a shared kernel object, I propose to extend the set of read-shared addresses R^p_τ respectively the set of write-shared addresses $W^p_{\tau'}$ in a similar way. More precisely, if the information-flow analysis of this call reveals an information flow from the input parameters into some shared kernel object, I propose to extend the set of addresses A with the addresses of a placeholder object that is large enough to hold the input parameters of p's invocation. If the concurrently executing thread τ is able to learn information about such an input parameter, the address of this parameter in the placeholder object is added to the set R^p_τ. This way, $\mathit{effects}_p$ correctly captures the propagation of this parameter. In the analysis of p, the actual code of the system call can thereby be replaced with a simple marshalling operation, which collects all input parameters and stores them into the placeholder object. The case where p reads from a shared kernel or server object works accordingly.

4.5.5. C++ to *Toy*

With sequential, parallel and indeterminately sequenced composition, the translation of the C++ low-level language constructs to *Toy* is straightforward. Consider for example the simple pointer program p in Figure 4.4. According to the C++ standard, the value computation $*$**a + b** and the address computation $*$**c** are both sequenced before the side effect $*$**c = $*$a + b**. The value computations $*$**a + b** and $*$**c** are unsequenced [PC09, § 5.17 pt 1]. Similarly, $*$**a** and **b** are sequenced before $*$**a + b** but otherwise unsequenced. Assuming that the result of $*$**a + b** can directly be stored in $*$**c**, the corresponding *Toy* program is given by Equation 4.8 respectively by the dependency graph on the right side in Figure 4.4.

$$\begin{aligned} p := (((n_{[b]} = b \parallel (n_{[a]} = a\, ; n_{[*a]} = *(n_{[a]})))\, ; n_{[+]} = n_{[*a]} + n_{[b]}) \parallel \\ (n_{[c]} = c\, ; n_{[*c]} = *(n_{[c]})))\, ; n_{[*c]} = n_{[+]} \end{aligned} \tag{4.8}$$

For a better readability, I have omitted all $e2s$ conversions, expression types t and markers γ. The result parameter n^t_τ of an expression e^t is denoted by $n_{[e]}$. For example, $n_{[+]}$ holds the result of $*$**a + b**.

[11] As before, I assume that a suitable timing-leak transformation eliminates all possible timing leaks

```
int *a;
int *c;
int  b;

*c = *a + b;
```

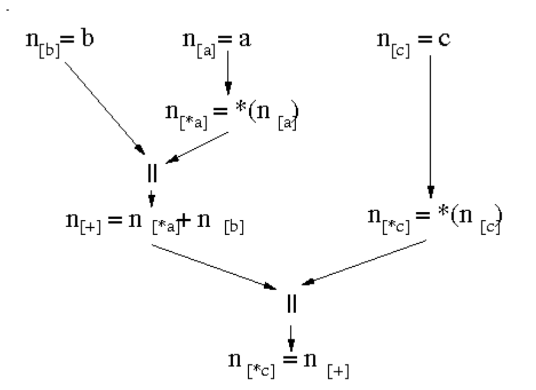

Figure 4.4.: A simple C++ pointer program (left) and its *Toy* translation (right). For better readability, the *Toy* program is shown as a dependency graph. The program executes from top to bottom, where horizontally neighbored instructions execute in an interleaved fashion.

To allow the compiler to optionally allocate the integer result $n_{[*c]}$ in the general purpose register eax, we have to adjust p as follows:

$$p; skip[](eax, 0) = n_{*c}$$

If the non-deterministic choice $[]$ evaluates to $skip$, the result remains only in $n_{[*c]}$. If it evaluates to $(eax, 0) = n_{*c}$, the integer result in $n_{[*c]}$ is stored to the address $(eax, 0)$, that is, to register eax at offset 0.

Because registers are explicitly contained in the type **Address** of the *Toy* memory model, the translation of inline assembler instructions to *Toy* programs is straightforward as long as jumps are trivial. GCC explicitly connects asm statements with the surrounding C++ program by specifying which registers and addresses hold the input and output parameters of asm statements. In addition, asm statements make explicit which of the non-output registers are potentially modified. These registers appear in the clobber list of the asm-statement. For example, the inline assembler statement in

```
bool Atomic::bit_test_and_clear(word & value, unsigned char bit) {

  bool result;

  asm volatile(" btr  %2, %1 "
               " setc %0       "
               : /* out */     "=a" ( result ),  "+m" (value)
               : /* in */      "b"  ( bit )
               : /* clobber */ "cc" );

  return result;
}
```

translates into

```
 // btr value, bit
n[addr] = &value + (EBX, 0) ;
n[c] = ∗(n[addr])^bit ;
( ∗(n[addr])^bit = 0 ||
  ∗(EFLAGS.c)^bit = n[c] ) ;

 // setc result
n[c]  = ∗(EFLAGS.c) ;
(EAX, 0) = n[c]
```

In Section 4.4, I assumed that jumps in the inline assembler statements of the to-be-checked operating-system code can trivially be replaced with corresponding if-statements or while-loops.

To translate C++ control-flow statements to *Toy*, we have to realize that **switch** can be expressed as a sequence of if-statements:

```
                                     if (c == 'a') {
  switch (c) {                         /∗ a ∗/
    case 'a' :                         ...
      /∗ a ∗/                          /∗ b ∗/
      ...                              ...
    case 'b' :            =          } else {
      /∗ b ∗/                          if (c == 'b') {
      ...                                /∗ b ∗/
      break;                             ...
    default :                          } else {
      /∗ default ∗/                      /∗ default ∗/
      ...                                ...
  };                                   }
                                     }
```

In this example, the code of the second case (**case 'b':**) is copied into the first case (**case 'a':**). This is to reflect that the first case does not terminate with a **break** statement. Similarly, **do** and **for** [PC09, § 6.5.3 pt 1] can be expressed as **while** loops:

```
  do { stmt; } while ( cond );        =     stmt; while( cond ) { stmt; }
```

and

```
  for ( init ; cond ; expr) { stmt; }      =     init ; while ( cond ) { stmt ; expr }
```

The C / C++ statements **break** [PC09, § 6.6.1], **continue** [PC09, § 6.6.2], and **return** [PC09, § 6.6.3] terminate loops respectively functions prematurely. Typically, these statements appear in the body of an if-statement. As a consequence, subsequent statements are not executed if the respective branch is taken. Information about the conditions that have lead to the execution of this branch can be leaked because premature terminations prevent the execution of externally visible side effects in the skipped statements. Unfortunately, subsequent statements are in general not at the same nesting level as the **break**. Hence, skipped code cannot always be placed into the else-branch of the if-statement that contains such a premature termination. The following program p demonstrates this points.

```
while (l) {
  if (l) {
    if (h) {
      break;
    }
    l = 5;
  }
  l = 7;
};
```

To correctly characterize the information flows due to premature termination, I introduced the special register **Prem**. Whenever a to-be-checked program contains a statement, which terminates parts of the program prematurely, the corresponding *Toy* program updates the register **Prem** with the cause of this termination. The translation places all code that follows this premature termination in the branch of if-statements, which check for the absence of this cause. As a result of this conversion, programs with premature terminations can be checked with a security type system that has no specific rules for these terminations. The implicit information flow is thereby characterized by the secrecy level of the cause in **Prem**. Applied to the above program p, the transformation produces the following code:

```
while (l && Prem != break) {
  if (l) {
    if (h) {
      Prem = break;
    }
    if (Prem != break)
      l = 5;
  }
  if (Prem != break)
    l = 7;
};
if (Prem == break)
  Prem = 0;
```

As mentioned at the beginning of Section 4.5, I assume that the translation inlines function calls and bodies.

4.6. Learned Secrets

Before we can turn into formalizing the security type system for *Toy*, we first have to understand which information concurrently executing threads can learn from the read-shared addresses and, consequently, which information they may return to the addresses they share with the to-be-checked program p in a writable fashion.

4.6.1. Secrets of the Initial State

Initially, before the to-be-checked system call or server invocation p starts, concurrently execution threads $\tau \in T$ can have learned confidential information from each other and placed this information into the writable addresses they share with p.

Definition 24. Initially Learned Secrets
Let T, W^p_τ, can_send^* be defined as in Section 4.5.4.1. Let further $l_0(\tau) \leq dom(\tau)$ be the least upper bound of the secrecy levels of information τ has accessed before p starts. I assume τ accesses any information in the directly or indirectly read-shared addresses of p. Then, L^0 denotes the secrecy levels of information that threads in T may have learned prior to the start of p and that they may have directly or indirectly written to the write-shared addresses of p. It holds that

$$\forall\, a \in \bigcup_{\tau \in T} W^p_\tau . L^0(a) = \bigsqcup_{\tau \in T_W(a)} l_0(\tau)$$

where $T_W(a) := \{\tau \in T \mid \exists\, \tau' \in T.\, \tau\ can\_send^*\ \tau' \wedge a \in W^p_{\tau'}\}$ is the set of threads that can potentially write to a. I call L^0 the learned secrets of the initial state before p starts executing. The learned secrets of non-write-shared addresses are undefined (i.e., the domain of L^i ranges over the addresses in $\bigcup_{\tau \in T} W^p_\tau$).

Based on the definition of the initially learned secrets, we can now define the secrecy levels of the initial typing environment M^0. It contains the secrecy levels of all addresses before p starts executing.

Definition 25. Initial Typing Environment
Let L^0 be the initially learned secrets (see Definition 24). Let further m_0 denote the initial secrecy levels of the addresses of p. Then, because I assumed threads to have accessed read-shared information of p, $m_0(a) \leq l_0(a')$ holds for read-shared addresses a with a effects$_p$ a' and

$$\forall\, a \in A. M^0(a) = \begin{cases} L^0(a) & \textbf{if } a \in \bigcup_{\tau \in T} W^p_\tau \\ m_0(a) & \textbf{otherwise} \end{cases}$$

holds for the typing environment of the initial state before p starts executing.

4.6.2. Evolution of Learned Secrets

Once p has started executing, the information at read-shared addresses and hence the learned secrets may change. If p writes confidential data to a non-local read-shared address, information about this data may propagate to all threads that have direct or indirect read access to this address and to all addresses of p that these threads can directly or indirectly write. To reflect this change, I maintain a second dynamic mapping of secrecy levels — the learned secrets L^i — in the control-flow-sensitive security type system, which I shall introduce in Section 4.7. The first dynamic mapping is the typing environment M^i.

Definition 26 contains the update rule for the learned secrets L^{i+1}. It is based on the learned secrets of the previous state L^i and on the typing environment M^{i+1}.

Definition 26. Update of Learned Secrets

Let L^i be the learned secrets L^i of the checked program p before p executes the atomic step i and let M^{i+1} be the typing environment after p has executes this atomic step i. Then, it holds for the learned secrets L^{i+1} that:

$$\forall\, a' \in \bigcup_{\tau \in T} W^p_\tau .\; L^{i+1}(a') = L^i(a') \sqcup \bigsqcup_{a \in R_{Eff}(a',i)} M^{i+1}(a)$$

In this equation, $R_{Eff}(a', i) := \{a \in \bigcup_{\tau \in T} R^p_\tau \mid a \notin local_p(i) \,\wedge\, a \; effects_p \; a'\}$ is the set of read-shared addresses a that are not lock protected during the atomic step i and that can effect the write-shared address a'.

After p has executed the i^{th} atomic step, a transition of concurrently executing threads follows. The secrets that these concurrently executing threads may learn from p is the data at non-local read-shared addresses. In addition, concurrently executing threads may remember previously learned secrets. Therefore, the secrecy level of the information these threads may propagate to a write-shared address a' is the least upper bound of the previously learned secrets $L^i(a')$ and of the new information in read-shared addresses a with a $effects_p$ a'. This leads to the update rule for the typing environment M^i, that is, to the typing rule for $\rightarrow_{ext(io)}$.

Definition 27. Typing Environment Update by Concurrently Executing Threads

Let A be the set of addresses, L^i the learned secrets for step i and M^i the typing environment that contains the secrecy levels of the previous step of the to-be-checked program p. Then, it holds for the typing environment M'^i that:

$$\forall\, a \in A.\; M'^i(a) = \begin{cases} M^i(a) \sqcup L^i(a) & \textbf{if } a \in \bigcup_{\tau \in T} W^p_\tau \,\wedge\, a \notin local_p(i) \\ M^i(a) & \textbf{otherwise} \end{cases}$$

M'^i reflects the secrecy levels of information concurrently executing threads could have stored into write-shared variables. It is the input typing environment of the atomic step $i+1$.

Given the typing environment M'^i, the typing rules of the security type system for *Toy*, which I shall present in Section 4.7, produce the typing environment M^{i+1}. Notice that the typing environment update rule for concurrently executing threads update the secrecy levels of non-local write-shared addresses in a weak fashion. That is, the update is with the least upper bound of both the old secrecy level $M^i(a)$ and the learned secret $L^i(a)$. This is to reflect that concurrently execution threads may also decide not to modify a in between the i^{th} and the $i+1^{st}$ atomic step of the to-be-checked program p.

Because local addresses can only be accessed by p, no secrets can be learned from local read-shared addresses and no learned secrets can be propagated into local write-shared addresses.

4.6.3. Constraining the Input Oracle to Produce l-Similar Inputs

To prove a security type system sound, we have to show for any observer that if the type system accepts a program p, then this program is non-interference secure. For the latter, we have to show that executing p on any two l-similar initial states s^0 and t^0 produces l-similar states provided that inputs are l-similar as well. The challenge here is that the learned secrets and thereby the decision whether two inputs are l-similar changes during the execution of p.

In [Völ08a], I characterized the inputs of concurrently executing threads with the help of three traces: two value traces, which contain arbitrary but fixed values; and one secrecy-level trace,

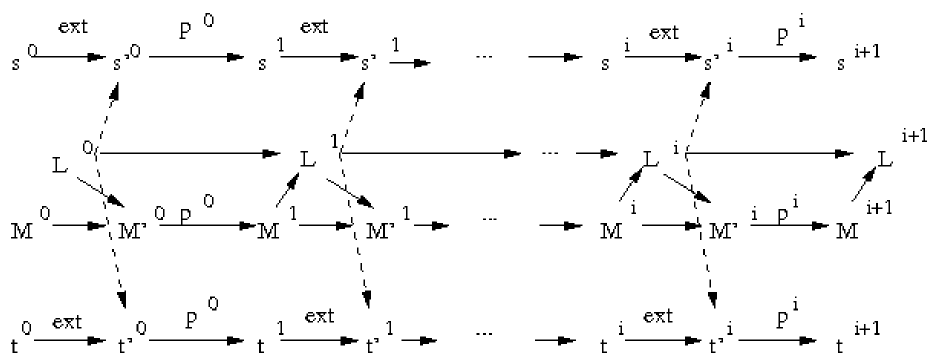

Figure 4.5.: Stepwise-interleaved evaluation of L^i, M^i, s^i and t^i.

which contains an upper bound on the secrecy levels of these values. If the secrecy level of an address at the position i in the trace is lower than or equal to l, the values for this address at i is set to be the same in the two value traces. I assumed the secrecy level of a at i to be lower than the learned secret $L^i(a)$. However, a formal connection was not given.

To formally connect input values with learned secrets, I execute, in this work, p with two input oracles io and io' on the two l-similar initial states s^0 respectively t^0. In the definition of shared-memory updates by concurrently executing threads (Definition 23), I have mentioned an input oracle, which is a function from observer secrecy level l, learned secret L^i and state s^i to a mapping $Address \rightarrow Bit \cup nil$. Having formally defined learned secrets, we can now complete this definition. io and io' are two functions of the above type, which are passed as parameters to $\rightarrow_{ext(io)}$ when p executes on s^0 respectively on t^0. The two functions io and io' are arbitrary but fixed except that they fulfil the following property:

Definition 28. l-similar Input Oracles

Let M^0 be the initial typing environment and let $s^0 \sim_{l,M^0} t^0$ indicate that the two initial states s^0 and t^0 are l-similar. Then the two input oracles io and io' must fulfil the following property:

$$\forall\, i \in \mathbb{N}, a \in \bigcup_{\tau \in T} W^p_\tau, s^i, t^i.$$
$$s^0 \overset{p}{\rightarrow}^i_c s^i \;\wedge\; t^0 \overset{p}{\rightarrow}^i_c t^i \;\wedge L^i(a) \leq l \Rightarrow io(l, L^i, s^i)(a) = io'(l, L^i, t^i)(a)$$

In the above equation, l is the secrecy level of the observer and $L^i(a)$ is the learned secrets for the atomic step i.

Definition 28 ensures that the two input oracles io and io' produce the same inputs whenever the learned secrets for a write-shared address a are lower than or equal to the observer secrecy level l.

The use of L^i in the above definition demands for a stepwise-interleaved evolution of L^i, M^i, s^i, and t^i. However, because s^i and t^i and hence the above evolution are required only for the soundness proof of the security type system, no overheads are imposed on an analysis with this type system.

Figure 4.5 illustrates this stepwise-interleaved evolution. Starting from $L^0(a) = M_0(a)$ for write-shared addresses a and from two l-similar initial states $s^0 \sim_{l,M^0} t^0$, the evolution of L^i proceeds in the following four steps:

- First, the typing environment M'^i is computed from M^i and from L^i using the typing environment update rule for concurrently executing threads (Definition 27);
- Then, the modifications of concurrently executing threads are propagated to the states s^i and t^i to produce s'^i and t'^i based on L^i (Definition 23);
- After that, the typing rule for the atomic step i advances M'^i to M^{i+1} and the semantics rule for this atomic step advances s'^i to s^{i+1} respectively t'^i to t^{i+1} with the input oracles io and io', which in turn depend on L^i; and,
- Finally, L^i is advanced to L^{i+1} using M^{i+1} (Definition 26).

Although the formalization of the concrete semantics of *Toy* relies on results from the security type system, it is well formed because the states s'^i and t'^i and hence also s^{i+1} and t^{i+1} rely only on results about the previous states: L^i, M^i, s^i, and t^i.

4.6.4. Example

To exemplify the use of learned secrets, let us return to the simple shared memory program p of Section 4.2.4.1 where **shm** is read-write shared with a $high$-classified program q, which so far has accessed only low-classified information:

```
1 tmp_a = shm;
2 shm = h;
3 shm = l;
4 tmp_b = shm;
5 l = tmp_a;
```

Is p secure with regards to a low-classified observer? Because q has so far only accessed low-classified information, the initially learned secrets of **shm** $L^0(shm)$ are low unless p would also share **h** with q in a read-shared fashion. Therefore, if we investigate the execution of p from two l-similar initial states s^0 and t^0 with $s^0`mem(h) \neq t^0`mem(h)$, $M^2(tmp\_a) = low$ and $s^0`mem(tmp\_a) = t^0`mem(tmp\_a)$ due to the constraint on the input oracles io and io'. Now, if p assigns **h** to **shm** in Step 4 (Line 2), $M^4(shm) = high$. As a result, the learned secrets $L^4(shm)$ are updated to high as described in the update rule in Definition 26. Although p resets **shm** to the low value of **l** in Step 6, $L^6(shm)$ remains $high$ to reflect that q may have remembered the previously assigned value of **h**. Hence, $M'^7(shm) = high$ to reflect that q may have returned information about **h** before p executes Step 7 in Line 4. The final typing environment M^{10} contains the following secrecy levels:

$$\begin{aligned} M^{10}(tmp\_a) &= low \\ M^{10}(l) &= low \\ M^{10}(shm) &= high \\ M^{10}(tmp\_b) &= high \end{aligned}$$

Hence, p is secure with regards to a low-classified observer because $M^i(l) \leq low$ for all $0 \leq i \leq 10$. However, if we would have assigned **tmp_b** to **l** instead of **tmp_a**, p would have to be rejected due to the possible leakage of **h** to **tmp_b**.

4.7. Security Type System for *Toy*

This section presents the control-flow-sensitive security type system for the deterministic core of *Toy* and its PVS-based soundness proof.

There are two principle approaches to cope with the non-determinism in *Toy*:

- We may extend the security type system for the deterministic core with the standard typing rules for non-deterministic composition [Sab01b, pg. 45]; or,
- We may check all possible ways in which the control-flow non-determinism in the to-be-checked program can be resolved, one way at a time.

In the next section, I shall elaborate on these alternatives. Section 4.7.2 presents the typing rules of the control-flow-sensitive security type system for *Toy*. In Section 4.7.3, I prove these rules sound against the formal semantics of *Toy*, which I have introduced in the previous section.

4.7.1. Control-Flow Non-Determinism

Obviously, the choice between checking control-flow non-determinism with the standard rules (e.g., Rule [ndet. choice] below) and checking all possible resolutions of this non-determinism involves a performance-precision tradeoff. To understand this tradeoff let us consider the following program p with a non-deterministic choice in Line 4.

```
1 int a = h;
2 int b = h;
3
4 int * c = (b = 0, &a) [] (a = 0, &b);
5
6 *c = 0;
7
8 l = a;
```

It sets both **a** and **b** to **0** independent of how the non-determinism in Line 4 is resolved. Hence, **l** reveals no secret information after **l = a**. p is secure with regards to low-classified observers.

Control-flow-sensitive security type systems typically include the following standard rule [Sab01b, pg. 45] to check non-deterministic choices of the form $c_1[]c_2$:

$$\text{[ndet. choice]} \quad \frac{[l_{ip}] \vdash M\ \{c_i\} M_i' \quad i \in \{1, 2\} \quad M' = M_1 \sqcup M_2}{[l_{ip}] \vdash M\ \{c_1[]c_2\}\ M'}$$

Applying this rule to the above program p, the results of typing **(b = 0, &a)** and **(a = 0, &b)** are merged into a single typing environment. As a consequence, the analysis of the remaining program $*$**c = 0; l = a;** needs to be carried out only once. It is much quicker than if we would check the two resolutions of the control-flow non-determinism in Line 4 separately. However, in the typing environment M_1 (for **(b = 0, &a)**), the secrecy level of **a** is $high$ and the secrecy level of **b** is low whereas in M_2 (for **(a = 0, &b)**), **a** is low and **b** is $high$. Hence, in the merged typing environment M', the secrecy levels of both **a** and **b** are $high$. The leads to the rejection of p because $*$**c = 0** in Line 6 cannot reduce $M'(a)$ to low. Because a correct points-to analysis cannot return more specific pointer targets than $*c \in \{a, b\}$, the typing rule for $*$**c = 0** must perform a weak update on $M'(a)$ and $M'(b)$. Therefore, the secrecy level of **a** is $high$ for the check of **l = a**.

The two ways in which the non-deterministic choice in Line 4 of the above program p can be resolved leads to the following two programs:

- **p1 := c = (b = 0, &a);** $*$**c = 0; l = a;** and
- **p2 := c = (a = 0, &b);** $*$**c = 0; l = a;**.

For the following two reasons, a separate information-flow analysis of these programs establishes the security of p. p is secure because

1. only one of the variables **a** and **b** assumes the secrecy level $high$ in the respective typing environment for the pointer assignment **c = (...)** ; and because
2. in both programs, the target of **c** can be determined precisely.

The costs of this precision double because two programs have to be checked instead of one.

4.7.1.1. Checking all Resolutions of Control-Flow Non-Determinism

Toy clearly separates control-flow non-determinism and input non-determinism. The former is introduced in the three *Toy* statements $c_1[]c_2$, $c_1 \diamond c_2$ and $c_1 \parallel c_2$ and resolved by the oracle $pick$; the latter is resolved by the oracle io, which we have already discussed above.
The oracle $pick$ takes a marker γ and returns either $true$ or $false$. The semantics of $c_1 \underset{\gamma}{[]} c_2$, $c_1 \underset{\gamma}{\diamond} c_2$ and $c_1 \underset{\gamma_0}{\parallel} c_2$ makes use of this decision to either favor the left statement c_1 or the right statement c_2. For a concrete instance of the oracle $pick$ we can therefore simplify the to-be-checked *Toy* program to a program that contains no control-flow non-determinism. $c_1 \underset{\gamma}{[]} c_2$ simplifies either to c_1 or to c_2, $c_1 \underset{\gamma}{\diamond} c_2$ simplifies either to $c_1; c_2$ or to $c_2; c_1$, and $c_1 \underset{\gamma_0}{\parallel} c_2$ simplifies to one concrete interleaving of the atomic steps in c_1 and c_2.
A sound analysis must check all resolutions of the non-determinism in these statements. For example, if the program contains $c_1 \underset{\gamma}{\diamond} c_2$, both programs must be checked: the one containing $c_1; c_2$ and the one containing $c_2; c_1$.

In some situations, a checked program p can be accepted as secure even if the check for some resolutions of the non-determinism in p fails. For example, assume $c_1; c_2$ succeeds for a program, which contains $c_1 \underset{\gamma}{\diamond} c_2$, whereas the simplified version of p, which contains $c_2; c_1$, is rejected as potentially being insecure. If this situation occurs in the translated C++ operating-system code, compilers that avoid such an unsafe resolution can still produce an information-flow secure binary from p. If this situation occurs in a hardware side effect, architectures that avoid such an unsafe resolution can still run the binary of p in an information-flow secure way. Because the translation from C++ to *Toy* typically introduces non-determinism to allow for subsequent compiler optimizations, a failure to check all possible resolutions of this non-determinism typically only prevents the corresponding optimization. In hardware side effects on the other hand, control-flow non-determinism is typically introduced to describe a hardware behavior whose implementation is typically not precisely known. A failure to check all possible resolutions of non-determinism in hardware side effects is therefore more critical.

4.7.1.2. Practicality

For the above example, the analysis worked because of the atomic steps that p executes, only one step involved a non-deterministic choice. In general, a program contains much more non-deterministic steps, which raises the question whether it is at all practical to check all possible resolutions of control-flow non-determinism.

In general, for an analysis of large programs with possibly non-terminating while-loops, the clear answer is "no". However, in our setting, we seek to check only terminating system calls and server invocations, which contain no true while-loops. Hence, the number of atomic steps is bounded and small and so is the number of atomic steps that affect the control flow of the to-be-checked program in a non-deterministic way.

Also, it is straightforward to extend the security type system for *Toy* with the standard rule for non-deterministic choice [Sab01b, pg. 45], which would allow us to apply the above approach only to selected non-deterministic steps. The development of heuristics for when to apply the standard rules and when to check all possible resolutions of a non-deterministic statement, is left for future work.

4.7.2. Typing Rules for the Deterministic Core of *Toy*

The typing judgements of *Toy* statements c (and likewise of *Toy* expressions) have the form:

$$[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{c\}\ M^{i+k},\ L^{i+k-1},\ i+k$$

Given a context secrecy level l_{ip} and the clearances M_c of potential observers of addresses $a \in A$, a statement c, which takes k atomic steps to execute, can be typed as follows. Starting from the typing environment M^i and the learned secrets [12] L^{i-1}, which correspond to the i^{th} atomic step of the to-be-checked program p (before the learned secrets have been updated with M^i), c can be typed if it modifies the addresses A such that their secrecy levels are those in M^{i+k} and the learned secrets are L^{i+k-1}. The security type system establishes that the program p can be typed as $[l_{ip},\ M_c] \vdash M^0, L^{-1}, 0\ \{p\} M^{|p|}, L^{|p|-1}, |p|$ where $L^{-1} = L^0$, then p is non-interference secure.

The typing rules of the control-flow sensitive type system for *Toy* have the following form:

$$\frac{(M^{i+k}, L^{i+k-1}) = AI(c)(M^i, L^{i-1}, i) \quad M^{i+k} \leq^{i+k} M_c}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{c\}\ M^{i+k},\ L^{i+k-1},\ i+k}$$

They consist of an abstract-interpretation part AI, which describes the update of the dynamic secrecy levels in the typing environment M^i and in the learned secrets L^i, and a clearance check $M^{i+k} \leq^{i+k} M_c$, which checks whether all secrecy levels of non-local addresses in the resulting typing environment M^{i+k} are dominated by the clearance secrecy levels M_c of potential observers of these addresses. Notice that a clearance check is contained in all typing

[12] The index reduction by 1 is a technicality, which is required to combine the update rule for learned secrets (Definition 26) and the update rule for the typing environment (Definition 27) into the same typing rule. See Figure 4.5.

rules. We shall return to this point in the machine-checked soundness proof in Section 4.7.3.3. The definition of $\leq^{i+k}$ is given below.

Similar to Definition 25, it must hold that:

Definition 29. Clearance of Read-Shared Variables

Like before, let T be the set of concurrently executing threads, let R^p_τ be the set of addresses that the to-be-checked program p shares in a readable fashion with $\tau \in T$, and let τ $can\_send^* \tau'$ hold if τ can directly or indirectly send to τ'. Then, for the clearance $M_c(a)$ of read shared addresses a, the following condition must hold:

$$\forall\, a \in \bigcup_{\tau \in T} R^p_\tau .\ M_c(a) = \bigsqcap_{\tau \in T_R(a)} dom(\tau)$$

In this definition, $T_R(a) := \{\tau \in T \mid \exists \tau' \in T. \tau'\ can\_send^*\ \tau\ \wedge\ a \in R^p_{\tau'}\}$ describes the set of threads that can directly or indirectly read the read-shared address a.

The constraint in Definition 29 ensures that the clearance $M_c(a)$ of a read-shared variable a is at most as high as the smallest secrecy level to which threads with direct or indirect read access to a are cleared. Clearly, the secrecy levels of initially-stored information must be dominated by the clearance M_c. That is, $M_0(a) \leq M_c(a)$ must hold for all addresses a.

Unless p holds a protecting lock, concurrently executing threads may observe read-shared addresses after any atomic step of p. Hence, we have to check every intermediate typing environment M^i to obey the clearance of non-local read-shared addresses [JPW05]. I write $M^i \leq^i M_c$ to abbreviate:

$$M^i \leq^i M_c := \forall\, a \in A.\ a \notin local(i)\ \wedge\ a \in \bigcup_{\tau \in T} R^p_\tau\ \Rightarrow M^i(a) \leq M_c(a) \tag{4.9}$$

Figure 4.6 shows the typing rules for *Toy* expressions, the typing rules for *Toy* statements are shown in Figure 4.7.

The typing rule $[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ exp\ \}\ M'^i,\ L^i,\ i+1$ combines the update rule for learned secrets (Definition 26), the update rule of write-shared addresses (Definition 27). In addition, it advances the atomic step count by 1. It is defined as follows:

Definition 30. Typing Rule for Concurrently Executing Threads

$$[ext]\ \frac{\begin{array}{lcl} L^i & = & \lambda a' \in \bigcup_{\tau \in T} W^p_\tau .\ L^{i-1}(a') \sqcup \bigsqcup_{a \in R_{Eff}(a',i)} M^i(a) \\ M'^i & = & \lambda a \in A.\ \begin{cases} M^i(a) \sqcup L^i(a) & \textbf{if } a \in \bigcup_{\tau \in T} W^p_\tau\ \wedge\ a \notin local_p(i) \\ M^i(a) & \textbf{otherwise} \end{cases} \end{array}}{[l_{ip}, M_c] \vdash M^i,\ L^{i-1},\ i\{\ ext\ \}\ M'^i,\ L^i,\ i+1}$$

In this typing rule, $R_{Eff}(a', i) := \{a \in \bigcup_{\tau \in T} R^p_\tau \mid a \notin local_p(i)\ \wedge\ a\ effects_p\ a'\}$ is the set of read-shared addresses a that are not lock protected during the atomic step i and that can effect the write-shared address a' (see Definition 26), W^p_τ and R^p_τ denote the read respectively the write-shared addresses with τ.

Similar to the formal semantics of *Toy*, I use $M^i\ [\ (a) \overset{t}{\mapsto} l\ ]$ and $read(t,a)(M^i)$ to denote updates respectively reads of the typing environment M^i: $M^i\ [\ (a) \overset{t}{\mapsto} l\ ]$ updates M^i at

$$\text{[const]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} l_{ip}\ ] \quad M^{i+1} \leq^{i+1} M_c}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ c^t(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[read]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} read(t,v)(M'^i) \sqcup l_{ip}\ ] \quad M^{i+1} \leq^{i+1} M_c}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ v^t(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[read ptr]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \qquad S = pta(i, n^{addr}) \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} \bigsqcup_{v \in S} read(t,v)(M'^i) \sqcup read(addr, n^{addr})(M'^i) \sqcup l_{ip}\ ]}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ *(n^{addr})^t\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[binary op]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} read(t,n_1^t)(M'^i) \sqcup read(t,n_2^t)(M'^i) \sqcup l_{ip}\ ]}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ n_1^t \circ n_2^t(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[unary op]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} read(t,n^t)(M'^i) \sqcup l_{ip}\ ]}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ \bullet n^t(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[cast op]}\quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \qquad M^{i+1} = M'^i\ [\ (n_r^t) \overset{t}{\mapsto} read(t',n^{t'})(M'^i) \sqcup l_{ip}\ ]}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ (t)n^{t'}(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

Figure 4.6.: Typing Rules for *Toy* Expressions

The typing rule for ext, which characterizes the dynamic secrecy level updates by concurrently executing threads, follows in Definition 30. Like in the transition rules of the *Toy* semantics, I use $M^i\ [\ (a) \overset{t}{\mapsto} l\ ]$ to denote updates of the typing environment M^i and $read(t,a)(M^i)$ to read the secrecy levels in M^i that are stored in the support bits of the type t. The relation $\leq^i$ checks whether secrecy levels are point-wise cleared (see Equation 4.9).

$$\text{[e2s]} \quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ e^t(n_r^t)\ \}\ M^{i+1},\ L^i,\ i+1}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ e2s(n_r^t = e^t)\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[skip]} \quad \frac{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} = M'^i \quad M^{i+1} \leq^{i+1} M_c}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ skip\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[write]} \quad \frac{\begin{array}{c}[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \\ M^{i+1} = M'^i\ [\ (v) \stackrel{t}{\mapsto} \bigsqcup read(t, n^t)(M'^i) \sqcup l_{ip}\ ]\end{array}}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ v = n^t\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[write str]} \quad \frac{\begin{array}{c}[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \\ S = pta(i, n^{addr}) \quad v \in S \quad |S| = 1 \\ M^{i+1} = M'^i\ [\ (v) \stackrel{t}{\mapsto} read(t, n^t)(M'^i) \sqcup read(addr, n^{addr})(M'^i) \sqcup l_{ip}\ ]\end{array}}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ *(n^{addr}) = n^t\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[write wk]} \quad \frac{\begin{array}{c}[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ ext\ \}\ M'^i,\ L^i,\ i+1 \quad M^{i+1} \leq^{i+1} M_c \\ S = pta(i, n^{addr}) \quad |S| > 1 \\ M^{i+1} = \lambda a. \begin{cases} \begin{array}{l} read(addr, n^{addr})(M'^i) \sqcup \\ \quad read(t, n^t)(M'^i) \sqcup l_{ip} \sqcup M'^i(a) \end{array} & \textbf{if } \exists\, v \in S.\ a - v \in support^t \\ M'(a) & \textbf{otherwise} \end{cases}\end{array}}{[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ *(n^{addr}) = n^t\ \}\ M^{i+1},\ L^i,\ i+1}$$

$$\text{[seq]} \quad \frac{\begin{array}{ll}[l_{ip}, M_c] \vdash & M^i, L^{i-1}, i\ \{c_1\} M^j, L^{j-1}, j \\ [l_{ip}, M_c] \vdash & M^j, L^{j-1}, j\ \{c_2\} M^k, L^{k-1}, k\end{array} \quad M^k \leq^k M_c}{[l_{ip}, M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ c_1; c_2\}\ M^k,\ L^{k-1},\ k}$$

$$\text{[if]} \quad \frac{\begin{array}{c}[l_{ip},\ M_c] \vdash M^i,\ L^{i-1},\ i\ \{\ \textbf{skip}\ \}\ M^{i+1},\ L^i,\ i+1 \\ [l'_{ip}, M_c] \vdash M^{i+1}, L^i, i+1\{\ c_1; skip_{k-|c_1|}\ \} M_1^{i+k+1}, L_1^{i+k}, i+k+1 \\ [l'_{ip}, M_c] \vdash M^{i+1}, L^i, i+1\{\ c_2; skip_{k-|c_2|}\ \} M_2^{i+k+1}, L_2^{i+k}, i+k+1 \\ k = max(|c_1|, |c_2|) \quad l'_{ip} = l_{ip} \sqcup read(bool, n^{bool})(M^{i+1}) \\ M^{i+k+1} = M_1^{i+k+1} \sqcup M_2^{i+k+1} \quad L^{i+k} = L_1^{i+k} \sqcup L_2^{i+k} \quad M^{i+1} \leq^{i+1} M_c\end{array}}{[l_{ip}, M_c] \vdash M^i,\ L^{i-1},\ i\{\ \textbf{if}(n^{bool})\{c_1\}\textbf{else}\{c_2\}\ \}M^{i+k+1}, L^{i+k}, i+k+1}$$

Figure 4.7.: Typing Rules for the deterministic *Toy* Statements

The typing rule for ext and the notions for typing environment updates and reads are as described in Figure 4.6. Equation 4.9 defines the relation $\leq^i$.

$a + k$ where $k \in support^t$ with the secrecy level l; $read(t, a)(M^i)$ reads all secrecy levels $M^i(a + k)$ where $k \in support^t$ and returns the least upper bound of these secrecy levels (i.e., $read(t, a)(M^i) = \bigsqcup_{k \in support^t} M^i(a + k)$).

Except that they require a typing of ext and except that all rules contain clearance checks, the typing rules for *Toy* expressions are standard. Notice, however, the write to the non-allocated temporary n_r^t and, in particular, the secrecy level l_{ip} that is considered for this write. Remember, l_{ip} is the secrecy level of the context in which the write is executed. The typing rule [read ptr] reaps benefit of the result of the correct points-to analysis to obtain the set S of possible targets for the pointer in n^{addr}. It reads the secrecy levels of all these addresses and returns the least upper bound of these secrecy levels, of the pointer, and of l_{ip}. The returned secrecy level is stored in the non-allocated temporary n_r^t.

With the exception of [write str], [write wk] and [if], the typing rules for statements are also standard. [write str] is the rule for writes through pointers, where the points-to analysis is able to precisely determine which address v the pointer targets. In this case, a strong update can replace the secrecy levels of the addresses $v + i$ where $i \in support^t$. The type t is the type of the written value.

[write wk] is the corresponding rule for writing though pointers whose target address the points-to-analysis cannot precisely determine. Because I assume the points-to analysis to be correct, the write modifies at most addresses in the set S, which the points-to analysis returns. Because we do not know which precise address $v \in S$ is modified, we must pessimistically assume that any such address keeps information about its old value. Therefore, [write wk] performs a weak update of the secrecy levels, which correspond to the support bits of any such address $v \in S$. That is, $M^{i+1}(v + i) = l \sqcup M'^i(v + u)$ where $l = read(t, n^t)(M'^i) \sqcup read(addr, n^{addr})(M'^i) \sqcup l_{ip}$ is the least upper bound of the secrecy levels of the stored information, of the pointer and of the context.

The rule [if] first skips one step to allow for preemptions after evaluating the condition n^{bool} and before the branches. After that, it requires that both branches are typed with a context secrecy level l'_{ip}, which is the least upper bound of l_{ip} and of the secrecy level of the condition. Note that the typing rules for the branches check the clearance for all typing environments M_1^x and M_2^x, $x \in \{i + 1, \ldots, i + k + 1\}$. However, actually $M_1^x \sqcup M_2^x \leq^x M_c$ must hold. Fortunately, for a lattice $(L, \leq)$ it holds that $a \leq c \wedge b \leq c \Leftrightarrow a \sqcup b \leq c$ for $a, b, c \in L$. Therefore, $M_1^x \sqcup M_2^x \leq^x M_c$ follows from the clearance checks of the respective branches.

Notice the lack of a typing rule for while. Although rules such as Rule C4 or Rule C4' of the control-flow-sensitive security type system in Figure 2.2 on page 28 are sound, such a typing rule is not required because all while-loops are assumed to terminate and because $while^i$ collapses to a sequence of if-statements.

The lack of a subsumption rule is intentional, although it is straightforward to show that such a rule (e.g., Rule S of Figure 2.2) is sound. As long as programs can interact with each other through shared memory, I don't expect much benefit of a subsumption because an application of this rule would only overestimate the possible information flows. Because all typing rules take two different typing environments and because the clearance M_c is provided in addition, a subsumption rule is not required.

In the accompanying PVS sources [Völ10], I slightly deviate from the formalization presented in Figure 4.6 and in Figure 4.7. Instead, I formalize the abstract-interpretation part of these typing rules in such a way that the individual typing environments M^i are returned for all atomic steps $i \in \{0, \ldots, |p|\}$ of the to-be-checked program p. This way, the clearance check can be separated form the computation of the corresponding typing environments. In particular, we can deduce the latter without requiring that p can be typed.

4.7.3. Soundness

This section presents the machine-checked soundness proof of the security type system for the deterministic core of *Toy*. Hence, I assume that all possible resolutions of non-deterministic choices are checked, however, only one at a time. It should be straightforward to also establish soundness for the standard typing rules for $[]$ and for $\|$. Following Warnier et al. [JPW05], I split the soundness proof of the proposed security type system into two parts:

- First, I show that the abstract-interpretation part of the security type system is *good* in the sense that if the program executes on l-similar initial states the resulting states are l-similar with respect to the dynamic typing environment M^i.
- Then, I show that goodness and the point-wise clearance $M^i \leq^i M_c$ (for all i) imply the desired non-interference property.

The non-interference property, which I will show, is termination and timing insensitive. That is, all checked system calls and server invocations must terminate to not leak information and a subsequent timing-leak transformation must be able to eliminate all remaining internal timing leaks.

4.7.3.1. Goodness

To prove the proposed security type system sound, I have to relate the formal semantics of *Toy* statements and expressions and the typing rules for these statements and expressions such that if a *Toy* program can be typed, it is non-interference secure. For that, each expression and statement must fulfil the following two properties:

1. Execution preserves l-similarity over dynamic types; and
2. If the checked program p modifies an address a, then the dynamic secrecy level of this address is at least as large as the context secrecy level l_{ip}.

More formally, let $\sim_{L\times[A\to L]} \subseteq \mathit{State} \times \mathit{State}$ be the l-similarity relation over concrete program states. l-similarity relates two states if their memories are observationally indistinguishable by an l-classified observer and if their atomic step counts match:

Definition 31. l-similarity over dynamic types
Two states s and t with $s`ip = t`ip$ are l-similar with regards to the dynamic secrecy levels in the typing environment M if they are related by the following relation over states:

$$s \sim_{l,M} t :\Leftrightarrow \forall a \in A.\ M(a) \leq l \Rightarrow s`mem(a) = t`mem(a)$$

The intuition behind this definition is that values may only differ if their dynamic secrecy level $M(a)$ is higher than or incomparable with l. In situation where l-similarity is only required for a subset $B \subseteq A$, I write $s \sim_{l,M|_B} t$ to mean

$$\forall a \in B.M(a) \leq l \Rightarrow s`mem(a) = t`mem(a)$$

Definition 32 formally defines the first property: Execution preserves l-similarity over dynamic types.

Definition 32. Execution preserves l-similarity
Assume the statement p is executed starting with step i from the two states s^i and t^i. Assume further that M^i is the typing environment that is provided to the typing rule for p and that M^k ($i \leq k \leq i + |p|$) is a typing environment that is produced by the abstract interpretation part $AI(p)$ during the typing of p (if p can be typed, such an M^k exists which appears on the right hand side of a typing judgement of a sub-statement or sub-expression of p). Then, the execution of p preserves l-similarity over dynamic types if the following condition holds:

$$\forall s^k, t^k.\ s^i \sim_{l,M^i} t^i \ \wedge\ s^i \overset{p}{\rightarrow}_c{}^{k-i}\ s^k \ \wedge\ t^i \overset{p}{\rightarrow}_c{}^{k-i}\ t^k \ \Rightarrow s^k \sim_{l,M^k} t^k$$

In this condition io and io' are two l-similar input oracles, which are used by $\overset{p}{\rightarrow}_c$ to execute p on s^i respectively on t^i. See Definition 28 for the definition of l-similar input oracles. In the above condition, $\overset{p}{\rightarrow}_c{}^n$ stands for the evaluation of the first n atomic steps of p.

It reads as follows: Given two states s^i and t^i that are l-similar with respect to the secrecy levels in the typing environment M^i, executing the statement p from these states up to the atomic step k produces two new states s^k and t^k that are l-similar as well with respect to the typing environment M^k that appeared during the derivation of p. Notice that because I require l-similar input oracles io and io' for the transition of p from s^i respectively from t^i, inputs to p are also l-similar.

In the formalization of the second property — if p modifies an address a, its secrecy level $M^k(a)$ is at least as large as l_{ip} — it is tempting to check inequality with the starting state s^i (see for example Warnier et al. [JPW05]):

$$\forall s^i, s^k.\ s^i \overset{p}{\rightarrow}{}^{k-i}\ s^k \ \wedge\ s^i\text{‘}mem(a) \neq s^k\text{‘}mem(a) \Rightarrow l_{ip} \leq M^k(a) \tag{4.10}$$

However, because p shares memory and other objects with concurrently executing threads, it can happen that these concurrently executing threads update s^k‘$mem(a)$ to the same value as s^i‘$mem(a)$. Obviously, in this case $l_{ip} \leq M^k(a)$ needs not to hold. To tolerate inputs of concurrently executing threads, I adjust Equation 4.10 to check inequality with the state s^k_{skip} that is obtained by executing only **skip** statements on s^i. In s^k_{skip}, only concurrently executing threads have modified memory addresses.

Definition 33. If p modifies an address a, $M^k(a)$ dominates l_{ip}
Assume k, l, l_{ip}, M^i, s^i, and M^k are as described in Definition 32 above. Then, the statement p fulfills the second property — if p modifies a, $M^k(a)$ dominates l_{ip} — if it holds that:

$$\forall s^k, s^k_{skip}, a.\ s^i \overset{p}{\rightarrow}_c{}^{k-i}\ s^k \ \wedge\ s^i \overset{\mathbf{skip}_{|p|}}{\rightarrow}_c{}^{k-i}\ s^k_{skip} \ \wedge\ s^k\text{‘}mem(a) \neq s^k_{skip}\text{‘}mem(a) \Rightarrow l_{ip} \leq M^k(a)$$

The primary purpose of the property in Definition 33 is to ensure that the typing rules correctly detect implicit information flows.

Definition 34. Goodness
A Toy *program p is good if it fulfils the properties of Definition 32 and of Definition 33:*

$$\begin{gathered}
good(p) :\Leftrightarrow \forall k, l, l_{ip}, M^i, M^k, L^{i-1}, L^{k-1}, s^i, t^i, s^k, t^k, s^k_{skip}, a. \\
(M^k, L^{k-1}) = AI(p)(M^i, L^{i-1})^{k-i} \ \wedge\ s^i \overset{p}{\rightarrow}_c{}^{k-i}\ s^k \ \wedge\ t^i \overset{p}{\rightarrow}_c{}^{k-i}\ t^k \ \wedge\ s^i \overset{\mathbf{skip}_{|p|}}{\rightarrow}_c{}^{k-i}\ s^k_{skip} \ \wedge \\
(s^i \sim_{l,M^i} t^i \ \Rightarrow\ s^k \sim_{l,M^k} t^k) \wedge \\
(s^k\text{‘}mem(a) \neq s^k_{skip}\text{‘}mem(a) \ \Rightarrow\ l_{ip} \leq M^k(a))
\end{gathered}$$

In this definition, $AI(p)(...)$ denotes the abstract-interpretation part of the typing rules for p. $AI(p)(...)^n$ stands for an abstract interpretation of the first n atomic steps of p.

The proof that all *Toy* programs are *good* is by structural induction over the expressions and statements of *Toy*. See Section 4.7.3.4 below.

4.7.3.2. Point-Wise Clearance

If a program p can be typed, it is easy to see that the following property holds for all typing environments that appear on the right hand side of the typing judgements for the sub-expressions and sub-statements of p (see Section 4.7.3.4 below).

Definition 35. Point-Wise Clearance

Assume M^i ranges over the typing environments that appear on the right-hand side of the typing judgements for the sub-expressions and sub-statements of p. Then, given M^0, M_c, L^{-1}, and l_{ip} as described before, the typing of p produces point-wise cleared secrecy levels if the following condition holds:

$$\begin{aligned} &\mathit{clearance}(p, l_{ip}, M^0, L^{-1}, M_c) := \\ &\qquad \forall i, M^i, L^{i-1}.\ (M^i, L^{i-1}) = AI(p)(M^0, L^{-1})^i \ \Rightarrow M^i \leq^i M_c \end{aligned}$$

Remember, for $M^i \leq^i M_c$ to hold it suffices that the secrecy levels of all non-local addresses a are dominated by $M_c(a)$ (see Equation 4.9).

4.7.3.3. Noninterference

A program p is non-interference secure with regard to an l-classified observer if this observer cannot distinguish any two runs of the program on any two states that vary in higher or incomparably-classified secrets or that receive varying higher or incomparably-classified inputs. Whenever a thread, which executes on behalf of such an observer can preempt p, it is able to directly or indirectly learn all information that is stored at read-shared non-local addresses a to which this thread τ is cleared (i.e., $M_c(a) \leq \mathit{dom}(\tau) \leq l$).

Hence, a program p is non-interference secure if p starts from any two l-similar initial states and if after any atomic step of p the resulting states are l-similar in the read-shared non-local addresses with $M_c(a) \leq l$. Definition 36 formalizes this property.

Definition 36. Confidential

Given an initial typing environment M^0 and the clearance M_c, the to-be-checked program p is non-interference secure if the following property holds for p:

$$\begin{aligned} &\mathit{confidential}(p, M^0, M_c) :\Leftrightarrow\ \forall i, l, s^0, t^0, s^i, t^i. \\ &\quad 0 \leq i \leq |p| \ \wedge\ M^0 \leq M_c \ \wedge\ s^0 \sim_{l,\, M^0} t^0 \ \wedge\ s^0 \xrightarrow{p}{}_c^{\,i} s^i \ \wedge\ t^0 \xrightarrow{p}{}^{\,i} t^i \ \Rightarrow\ s^i \sim_{l, M_c|_{RL^i}} t^i \end{aligned}$$

In this definition, $RL^i := \{a \in A |\ a \in \bigcup_{\tau \in T} R^p_\tau \ \wedge\ a \notin \mathit{local}(i)\}$ denotes the set of non-local read-shared addresses. Like before, T is the set of concurrently executing threads, R^p_τ is the set of read-shared addresses with $\tau \in T$ and local is as defined in Definition 22 on page 146.

Relation to Noninfluence: *confidential* is a simplified form of Noninfluence (see page 24): Let me first focus on the right-hand side of Definition 6: $\exists t^j \in S.\ t^0 \xrightarrow{\beta} t^j \wedge \mathit{output}(l, s^i) = \mathit{output}(l, t^j)$. Because the security type system for *Toy* is only for the deterministic core, the checked program p evaluates after i steps to precisely one state (i.e., s^i when p is executed on s respectively t^i when executed on t). Because we assume that p

terminates, we can use a termination-insensitive form of non-interference. In these properties, t^i appears as an universally quantified parameter. A proof of the existence of a suitable t^i is not required.

Assuming that hardware-centric covert channels have been mitigated, l-classified observers may learn information about p's execution only from concurrently-execution threads. The mapping of kernel- or server-object invocations to shared-memory addresses (Section 4.5.4.4) ensures that these threads can only learn information about p's execution from non-local read-shared addresses RL^i. Hence, $output(l, s^i) = output(l, t^i)$ in Definition 6 becomes $s^i \sim_{l,M_c|_{RL^i}} t^i$.

The second precondition — $s^0 \overset{sources(\alpha,l)}{\approx} t^0$ — on the left-hand side of Definition 6 simplifies to $s^0 \sim_{l,\ M_0} t^0$. The constraint on the input oracles io and io' (Definition 28) replaces the first precondition: $ipurge(\alpha, l) = ipurge(\beta, l)$. The fundamental difference is that the learned secrets evolve dynamically with the execution of p. Remember, for transitive information-flow policies $(L, \leq, dom)$, $sources(\alpha, l) := \{w | dom(a) = w \ \wedge \ w \leq l\}$.

4.7.3.4. Soundness Proof

To prove the security type system for *Toy* sound, I have to show for all programs p that if p can be typed, p is confidential:

Theorem 1. Soundness

The type system is sound. More precisely, for all p, M^0, $L^{-1} = L^0$, M_c, l_{ip}, l, *constrained as described above, if* $[l_{ip}, M_c] \vdash M^0, L^{-1}, 0\{p\}M^{|p|}, L^{|p|-1}, |p|$ *can be derived for a suitable* $M^{|p|}$ *and* $L^{|p|-1}$ *with the typing rules of the* Toy *security type system, then*

$$confidential(p, M^0, M_c)$$

holds.

Proof: The proof of Theorem 1 follows from the following Theorem and from Proposition 2, which states that if $[l_{ip}, M_c] \vdash M^0, L^{-1}, 0\{p\}M^{|p|}, L^{|p|-1}, |p|$ can be derived, it follows that $clearance(p, l_{ip}, M^0, L^0, M_c)$ holds. **q.e.d.**

Theorem 2. Main Theorem

$$\forall p, l_{ip}, M^0, L^0, M_c.\ good(p) \ \wedge \ clearance(p, l_{ip}, M^0, L^0, M_c) \ \Rightarrow confidential(p, M^0, M_c)$$

Proof: Choose, M^0, L^0, M_c, l, l_{ip}, s^0 and t^0 such that

1. observers are cleared to the initially stored secrets: $M^0 \leq M_c$,
2. s^0 and t^0 are l-similar (i.e., $s^0 \sim_{l,M^0} t^0$; see Definition 31), and
3. in some step i before p terminates[13] $s^i`mem(a) \neq t^i`mem(a)$ for some read-shared non-local address a (i.e., $a \in RL^i$ of Definition 36) where $s^0 \xrightarrow{p}_c{}^i s^i$, $t^0 \xrightarrow{p}_c{}^i t^i$.

[13] Remember, we implicitly assumed that p terminates eventually.

Then, we have to show that $M_c(a) \not\leq l$ in order to satisfy $s^i \sim_{l,M_c} t^i$ and hence confidentiality. Goodness (Lemma 1) gives us that $M^i(a) \not\leq l$ because $s^i \sim_{l,M^i} t^i$ would otherwise imply that s^i'$mem(a) = t^i$'$mem(a)$. From the clearance check $clearance(p, l_{ip}, M^0, L^0, M_c)$ we know that $M^i(a) \leq M_c(a)$ because $M^i \leq^i M_c$ and both $a \in \bigcup_{\tau \in T} R^p_\tau$ and $a \notin local(i)$. But then, the transitivity of $\leq$ leads to the desired result:

$$M^i(a) \leq M_c(a) \wedge M^i(a) \not\leq l \;\Rightarrow\; M_c(a) \not\leq l \;\Leftrightarrow\; M^i(a) \leq M_c(a) \wedge M_c(a) \leq l \;\Rightarrow\; M^i(a) \leq l$$

q.e.d.

Proposition 2. p **Obeys the Clearance of Addresses**
If $[l_{ip}, M_c] \vdash M^0, L^{-1}, 0\{p\}M^{|p|}, L^{|p|-1}, |p|$ *can be derived for a suitable* $M^{|p|}$ *and* $L^{|p|-1}$ *with the typing rules of the* Toy *security type system, then*

$$clearance(p, l_{ip}, M^0, L^0, M_c)$$

holds.

Proof: The proof follows trivially by realizing that all typing rules except [if] contain a clearance check $M^{i+1} \leq^{i+1} M_c$ for their respective result typing environment M^{i+1}. The typing rule [if] checks clearance of the **skip** step, which is used to allow for preemptions between the check and the branches. Let k range over the atomic steps of the branches. The typing rules for the branches perform clearance checks for all typing environments $M_1^k \leq^k M_c$ (if-branch) and for all typing environments $M_2^k \leq^k M_c$ (else-branch). The clearance check for $M^k = M_1^k \sqcup M_2^k$ follows from the well known result about lattices that $a \leq u \wedge b \leq u \Leftrightarrow a \sqcup b \leq u$. **q.e.d.**

Lemma 1. All Statements are Good

$$\forall p.\ good(p)$$

Proof: The proof proceeds by structural induction over p. The second clause of $good(p)$ — if p modifies a, $l_{ip} \leq M^k(a)$ (see Definition 33) — follows trivially by realizing that whenever the transition rule for statements $\rightarrow_c$ modify an address, the corresponding typing rule updates the secrecy level of the same address with a secrecy level, which is at least as large as l_{ip}.

The first clause of $good(p)$ — execution preserves l-similarity (see Definition 32) — follows straightforwardly if we realize that whenever $\rightarrow_c$ modifies an address, the corresponding typing rule updates the secrecy level of this address with a secrecy level that combines all secrecy levels of the parameters that contributed to the stored result. The proof of the individual statements is by case distinction.

Let us here focus only on the most interesting case: an if-statement with a higher-classified conditional. Because the conditional n^{bool} is higher-classified (e.g., at h) than the observer secrecy level l, we cannot deduce from $s^i \sim_{l,M^i} t^i$ that the if-statement executes the same branch in s^i and in t^i. To prevent the observer from deducing information about the conditional, we must then show that the secrecy level of any address that is modified in either branch is greater than l. From the first condition of $good$, we know that $l_{ip} \leq M^k(a)$ holds for all addresses a that have been modified by p. But since the branches are typed with a context secrecy level that is at least as high as h, it holds that $M^k(a) \not\leq l$, which leads to the desired l-similarity result

$s^k \sim_{l,M^k} t^k$. The case where an l-classified observer is cleared to see the conditional, follows trivially from the induction hypothesis if we realize that the same branch is executed. **q.e.d.**

For more details about this proof, the interested reader is referred to the published PVS sources [Völ10].

This concludes the soundness proof of the security type system for the deterministic core of *Toy*. I have machine-checked the above proofs with the help of the theorem prover PVS. The PVS sources are publicly available [Völ10].

4.8. Summary

In this chapter, I have identified several challenges that have to be addresses by security type systems for the low-level operating-system code of microkernel-based systems. These challenges origin from the peculiar ways in which the microkernel and the necessarily-trusted multi-level servers interact with their clients, with the underlying kernel and with the underlying hardware platform. To address these challenges, I have introduced the non-deterministic intermediate programming language *Toy*. The non-deterministic constructs of this language make it easy to translate the low-level C++ operating-system code into a *Toy* program and to describe the side effects from these peculiar interactions as interleaved-executing subprograms.

To prove data confidentiality of low-level operating-system code, I have introduced a control-flow-sensitive security type system for the deterministic core of *Toy*. The use of a universal lattice for shared-memory programs allows programs to be analyzed whose information-flow policy is not completely known at the time of the analysis. The analysis is protection parametric in the sense that certain invocations can be checked separately and with placeholder objects to compensate for unknown capability targets. In the special setting of low-level operating-system code, it is practicable to check, one at a time, all possible resolutions of the control-flow non-determinism in the corresponding *Toy* program.

I have used the theorem prover PVS to formalize the semantics of *Toy* and the typing rules of the *Toy* security type system and to machine check the soundness proof of this security type system.

In the next chapter, I demonstrate the applicability of the proposed information-flow analysis with the help of three case studies. In addition, I demonstrate the effectiveness of a countermeasure against AES cache side-channel attacks.

5. Case Studies

In this chapter, I demonstrate the information-flow analysis of Chapter 4 with three case studies: a virtual-memory access (Section 5.1), the IPC-send operation of an L4-family microkernel (Section 5.2), and a supposedly secure buffer-cache implementation (Section 5.3). In the first two case studies, I exemplify the analysis of hardware side effects (Section 4.2.2) respectively the protection-parametric analysis (Section 4.2.5) of a system call. The buffer-cache case study combines these results with the results of Chapter 3 about information-flow secure scheduling and synchronization.

In addition to these three case studies, I prove correctness of Osvik's countermeasure for AES. That is, this countermeasure effectively protects the key, the plaintext and the intermediate encryption results against cache side-channel attacks (Section 5.4). To my best knowledge, this is the first security-type-system-based proof of such a countermeasure. For want of a type-checking tool for *Toy*, I have crafted this proof by hand.

5.1. Page-Table Walk

In Section 4.3, we have already seen that virtual-memory reads contain implicit information flows into the accessed bits of used page-table entries. In this section, I demonstrate with the help of the following program p how the *Toy* security type system checks this hardware side effect to identify these information flows. Given a concrete information-flow policy, the produced results can be used to prove this side effect and the triggering program termination- and timing-insensitive non-interference secure.

```
1  register int h asm("eax");
2
3  int l __attribute__((aligned(sizeof<int>)));
4
5  if (h == 0) {
6    h = l;
7  }
```

In this case study, I assume that p executes on an IA32 processor with 32-bit paging [Cor09, § 4.3 Vol. 3a], 4KB and 4MB pages and flat segments (i.e., segments span the entire address space). I further assume that all accesses of p are to physical-memory regions with no hardware side effects. That is, the reads and writes of p target DMA-inaccessible RAM. In particular, they do not target memory-mapped device registers.

The storage-class specifier **register** and the **asm** declaration in Line 1 are hints to the C++ compiler of the GNU Compiler Collection (GCC) to allocate **h** in the general-purpose register **EAX** [SC08b, § 6.42]. Let us assume that GCC follows this hint.

Based on these assumptions, the C++ to *Toy* translation results in the program:

1 $((n_h = (EAX, 0)^{int} \;\|\; n_0 = 0^{int}); n_{==} = n_h == n_0);$
2 **if** $(n_{==})$ {
3 $\quad n_l = *(n^{addr}_{phys(l)})^{int}; (EAX, 0) = n_l$
4 } **else** {
5 $\quad$ skip
6 }

The temporary n_h holds the value of **h**, which is read from $(EAX, 0)$. n_l holds the value read from $n_{phys(l)}$, the physical address of **l**. To obtain this address, p has to invoke the hardware side effect **pte_walk(l)**. Figure 5.1 shows the *Toy* code for this hardware side effect [1].

In **pte_walk(l)**, let **a.x** denote an access to the field **x** of the bitfield **a**. I write $\mathbf{a}_{[31..12]}$ for the bit field of **a** that is stored in the bits 31 . . . 12. The operator $\circ$ stands for the bit-wise concatenation of two bitfields. For example, $n_{phys(l)} = n_{pde\ [31..22]} \circ virt(l)_{[21..0]}$ concatenates the upper 10 bits of n_{pde} with the lower 22 bits of $virt(l)$ to form the physical address $n_{phys(l)}$. The statement **if (exp) ...** abbreviates n_{if} **= exp; if** (n_{if}) **...** .

The hardware side effect **pte_walk(l)** starts in Line 4 with the extraction of the current privilege level **CS.hidden_dpl** from the hidden part of the code-segment register **CS**. For user-mode accesses this privilege level is 3. The address of the page directory (i.e., the first-level page table) is stored in the page-table base register **CR3**. The page-directory entry that is relevant for translating the virtual address $virt(l)$ can be obtained by indexing into this page directory with the 10 upper-most bits of this address. Because each page-table entry is 4 bytes large, the address of the page-directory entry that is used for the translation is: $\mathbf{CR3}_{[31..12]} \circ (virt(l)_{[31..22]} << 2)$. Notice, the evaluation order of this address computation and of the **CS** access is undefined. If the page-directory entry is present and if it conveys sufficient privileges for the access, the analysis proceeds by checking the page-size flag to distinguish a page-directory entry for a 4MB page from a page-directory entry that refers to a second-level page table. A page-directory (or page-table) entry conveys sufficient privileges for a user-mode read access if the present and the user flag of this entry are set. For kernel-mode read accesses only the first flag must be set. If any of the checks in Line 9 or in Line 21 fail, the hardware side effect raises a page-fault exception. To simplify the above code snippet, I make use of a signalled NaT to propagate the exception that has to be triggered after writing the fault information to the control register **CR2**. This allows the fault to be handled in a second hardware side effect. If the page-directory entry refers to a second-level page table, the translation proceeds in a similar way with the next lower 10 bits of the virtual address $virt(l)$. Once a page-table entry is found that refers to a page, the translation stops by concatenating the page base address in the page-table entry with the lower part of $virt(l)$ to obtain the physical address.

Because p reads **l** only if **h == 0** and because the translation result must be present before **l** is read, **pte_walk(l)** must be inserted before Line 3 in the above *Toy* program of p. If p would have computed the address of **h** or if p would have executed another expression before Line 3 that does not depend on **l**, the page-table walk would be unsequenced to these computations.

[1] If the operating-system kernel fails to properly invalidate the translation lookaside buffer and the paging-structure caches after page-table modifications, the translation of the MMU can deviate from the information in the page-table entries. For reasons of simplicity I omit these inconsistencies in the following *Toy* program **pte_walk**.

```
pte_walk(l) {

  ( // read current privilege level
    n_user = (CS.hidden_dpl == 3) ||

    // read cr3
    n_pde = *(CR3[31..12] o (virt(l)[31..22] << 2))) ;

  if (n_pde.present ∧ (n_user ⇒ n_pde.U/S)) {

    if (!n_pde.accessed) { n_pde.accessed = 1 } ||

    if (n_pde.ps) {

      n_phys(l) = n_pde[31..22] o virt(l)[21..0]

    } else {

      n_pte = *(n_pde[31..12] o (virt(l)[21..12] << 2)) ;

      if (n_pte.present ∧ (n_user ⇒ n_pte.U/S)) {

        if (!n_pte.accessed) { n_pte.accessed = 1 } ||

        n_phys(l) = n_pte[31..12] o virt(l)[11..0]

      } else {
        // signal page fault by returning an SNAT
        CR2 = set_fault_address_and_bits(virt(l), 0 /* read */, n_pte.present, CS.hidden_dpl != 3) ;
        n_phys(l) = SNAT(#PF)
      }
    }
  } else {
    // signal page fault by returning an SNAT
    CR2 = set_fault_address_and_bits(virt(l), 0 /* read */, n_pte.present, CS.hidden_dpl != 3) ;
    n_phys(l) = SNAT(#PF)
  }
}
// check SNAT
if (is_SNAT(n_phys(l)) {
  // trigger page-fault exception
  ...
}
```

Figure 5.1.: *Toy* program of the IA32 page-table walk hardware side effect.

An application of the *Toy* type-checking rules [read ptr], [write ptr] and [if] reveals the implicit flow of **h** into the accessed bits: **pte_walk(l)** is executed in a branch of an if-statement with $high$ conditional. The context secrecy level l_{ip} of the write to $\mathbf{n}_{pde}$**.accessed** is at least $high$. $\mathbf{n}_{pde}$**.accessed = 1** constitutes a write through the pointer in $\mathbf{n}_{pde}$. Hence, $M'(\mathbf{n}_{pde}\textbf{.accessed}) = M(\mathbf{n}_{pde}) \sqcup l_{ip} \sqcup \bot$. holds for the secrecy level of the accessed bit in the result typing environment M'. Because l_{ip} is at least $high$, the secrecy level of this accessed bit is also at least $high$.

In addition to this implicit flow, the analysis reveals also the variables that contribute to the secrecy level of the translation result and to the secrecy level of the set accessed bit. These are, the current privilege level **CS.hidden_dpl**, the present and user flags of the page-table entries and the pointers to the next page-table level respectively to the page-base address. The accessed bits and other information in the page-table entries have no influence on the translation result and hence on the read access.

For an analysis of the remaining steps of p, the abstract physical address $\mathbf{n}_{phys}$ can either be obtained from a points-to analysis, which evaluates the pointers in the page-table entries, or from the function $v2p$, which abbreviates this translation.

Having identified this information flow as a possible leak, the obvious question is how to avoid it. Several approaches are imaginable including the following:

1. Disable the setting of accessed bits in the hardware side effect by initializing all page tables with set accessed and dirty bits;

2. Avoid a leakage of accessed bits outside the kernel; or

3. Equip memory capabilities with an additional access right that authorizes the retrieval of accessed and dirty bits.

5.2. IPC

In this section, I demonstrate how the proposed information-flow analysis can be applied to identify potentially-harmful information flows in the IPC send operation of an L4-family microkernel. Because the system-call invoking user-level program is not known at the time of the analysis, the analysis must be parametric. Instead of a concrete information-flow policy $(L, \leq, dom)$, we use the universal lattice for shared-memory programs (see Section 4.2.4.1). A later instantiation of this lattice with the secrecy levels in L reveals whether the identified information flows are benign or whether they violate this policy.

The C++ source code in Figure 5.2 and in Figure 5.3 is a slightly-modified excerpt of the IPC path of the Nova Microhypervisor [Ste09b] by Udo Steinberg [2]. For a better readability, I will not translate this code into a corresponding *Toy* program or consider hardware side effects during this analysis. Figure 5.2 shows the kernel entry, kernel exit, and system-call dispatch code. Whenever a user-level program invokes a Nova system call with the **IA32_SYSENTER** instruction, the processor activates **entry_sysenter** with the kernel stack pointer set to the variable **Tss:run.sp0**. The kernel entry obtains the addresses of **entry_sysenter** and of **Tss:run.sp0** from the model specific registers **IA32_SYSENTER_EIP** and **IA32_SYSENTER_ESP**. These registers are two of the special-purpose registers contained in the type **Register_ID**.

[2] The Nova source code and hence this excerpt are released under the terms of the GNU General Public License Version 2 [Fou91].

```
entry_sysenter:         pop     %esp
                        lea     -44(%esp), %esp
                        pusha
                        mov     KERNEL_STACK_END, %esp
                        jmp     syscall_handler

void Thread::syscall_handler (uint8 number) __attribute__((regparm(1)));

void Thread::syscall_handler (uint8 number)
{
    if (EXPECT_TRUE (number == Sys_regs::MSG_CALL))
        sys_ipc_call ();
    if (EXPECT_TRUE (number == Sys_regs::MSG_REPLY))
        sys_ipc_reply ();
    ...

    sys_finish (&current->regs, Sys_regs::BAD_SYS);
}


void Thread::sys_finish (Sys_regs *param, Sys_regs::Status status)
{
    param->set_status (status);

    ret_user_sysexit ();
}

void Thread::ret_user_sysexit ()
{
    asm volatile ("lea %0, %%esp;"
                  "popa;"
                  "sti;"
                  "sysexit"
                  : : "m" (current->regs) : "memory");
    UNREACHED;
}
```

Figure 5.2.: Kernel entry and exit path for system calls of the Nova Microhypervisor

The variable **TSS:run.sp0** of the task-state segment stores a pointer that refers to the address immediately following the register safe area of the current thread. The first five assembler instructions of **entry_sysenter** safe the user registers in this area, activate the kernel stack, and invoke the C++ function **syscall_handler**. The first register parameter of this function is located in the EAX register (**REGPARM(1)**). The lower-most eight bits of this parameter encode the **hypercall number** (i.e., the opcode of the invoked system call).

When activated, the function **syscall_handler** checks this opcode and invokes the respective C++ function for the system call. Invalid system-call numbers cause an immediate return with **BAD_SYS** as status code. The function **sys_finish** respectively **ret_user_sysexit** complete the system call. The latter restores the registers of the user-level program from the register safe area and exits the kernel with the **IA32 SYSEXIT** instruction.

Because SYSEXIT returns to user level, it prematurely terminates the system call. Hence, the **UNREACHED** statement in **ret_user_sysexit** and any code that follows this statement in the calling functions will not be reached. To check the implicit information flows that occur when SYSEXIT is invoked in a branch of an if-statement, **SYSEXIT** must be treated like a return statement, which returns to the end of **syscall_handler**. Unlike the C++ statement **return**, and unlike exceptions, a premature termination through **SYSEXIT** does not execute the destructors of objects with automatic storage durations [PC09, § 12.4 pt 8, § 15.3 pt 11]. The corresponding *Toy* translation must therefore reset this premature termination in **PREM** only at the end of **syscall_handler** after skipping over all these destructors. Like the other premature terminations, **SYSEXIT** leaks information about the invoking context to all subsequent statements.

An analysis of Nova system calls for potentially harmful information flows requires at least three placeholder objects:

- the kernel stack;
- the thread control block of the system-call invoking thread, which includes the register safe area of this thread; and
- the CPU-local variable **current**, which refers to this thread.

In Nova, CPU-local variables have the same virtual addresses on all CPUs. However, the virtual-to-physical mappings of these addresses differ on different CPUs. Further placeholder objects for the analysis are therefore those parts of the per-CPU page tables that are used to translate the accessed CPU-local and global addresses.

As we have seen in Section 4.2.5, analyses of **syscall_handler**, which are not parametric in the system-call opcode, are not very interesting: an application-level program will never invoke all system calls at once. By checking all system calls individually in a parametric analysis, the analysis reveals the information flows of every system call in isolation. The results of such an analysis can then be used for example to check the invoking application-level program. To check system calls individually, we have to regard **number** as a parameter of the analysis. Obviously, the default case needs only to be checked once. An immediate consequence of turning **number** into a fixed parameter is that **syscall_handler** simplifies to the invocation of a single C++ function. Still, because **number** is a user-provided parameter in the conditional of the if-statement, which selects the C++ function, its secrecy level contributes to the context secrecy level l_{ip}.

Figure 5.3 shows the function **sys_ipc_call**, which is invoked by **syscall_handler** if the invoking thread has set the hypercall number in **EAX** to **Sys_regs::MSG_CALL**. In Nova, IPC call and IPC send share the same code. The only difference is that IPC send terminates the receive phase (**recv_ipc_msg**) prematurely.

Let us here focus on a send operation with send timeout zero and no capability transfers. That is, the bits **Sys_ipc_send::TIMEOUT_ZERO** and **Sys_ipc_send::SEND_ONLY** in the parameter **flags** and the number of typed items (**s**→ **mtd().typed()**) are further parameters of the analysis. These parameters are located in the message-transfer descriptor **s**→ **mtd()**. For the above send operation, the two flags have to be set and no typed items must be specified.

```
void Thread::sys_ipc_call () {

    Sys_ipc_send *s = static_cast<Sys_ipc_send *>(&current->regs);

    Capability cap = reinterpret_cast<Capability *>(OBJSP_SADDR)[s->pt() % max_caps];

    Kobject *obj = cap.obj ();
    if (EXPECT_FALSE (obj->type() != Kobject::PT))
        sys_finish (s, Sys_regs::BAD_CAP);

    Portal *portal = static_cast<Portal *>(obj);
    Thread *receiver = portal->receiver;

    if (EXPECT_FALSE (current->cpu != receiver->cpu))
        sys_finish (s, Sys_regs::BAD_CPU);

    if (EXPECT_FALSE (!Atomic::bit_test_and_clear(receiver->wait, 0))) {
        if (EXPECT_FALSE (s->flags() & Sys_ipc_send::TIMEOUT_ZERO))
            sys_finish (s, Sys_regs::TIMEOUT);
        // receiver is not yet ready to receive; switch to receive
        ...
    }

    // transfer message
    receiver->utcb->mtd = Message_Transfer_Descriptor (s->mtd().untyped());

    for (unsigned long i = 0; i < s->mtd().untyped(); i++)
        receiver->utcb->mr[i] = mr[i];

    if (EXPECT_FALSE (s->mtd().typed()))
        // transfer capabilities
        ...

    current->continuation = ret_user_sysexit;

    receiver->utcb->portal_id = portal->node.base;

    // receive message
    receiver->recv_ipc_msg (portal->ip, s->flags());
}

void Thread::recv_ipc_msg (mword ip, unsigned flags)
{
    Sys_ipc_recv *r = static_cast<Sys_ipc_recv *>(&regs);

    r->set_ip (ip);

    if (EXPECT_FALSE (flags & Sys_ipc_send::SEND_ONLY)) {
        ready_enqueue();
        ret_user_sysexit ();
    }

    ...
}
```

Figure 5.3.: Source code of the IPC call operation of the Nova Microhypervisor

L4-IPC send proceeds in four phases:

1. parameter extraction and capability lookup (Lines 1 – 16),
2. rendezvous (Lines 17 – 23),
3. message transfer (Lines 24 – 33), and
4. preparation of output parameters and system-call exit (Lines 34 – 50).

The static cast of the register safe area **regs** to a variable of type **class Sys_ipc_send** (Line 3) provides a convenient interface to access the register-passed parameters of the IPC system call. These are the capability selector for the portal capability **s**$\rightarrow$ **pt()**, the hypercall flags **s**$\rightarrow$ **flags()**, and the message-transfer descriptor **s**$\rightarrow$ **mtd()**. The latter contains the number of untyped message words **s**$\rightarrow$ **mtd().untyped()**. If the IPC is successful, the kernel copies these untyped message words from the sender UTCB to the receiver UTCB. The message-transfer descriptor contains also the number of typed items, which we assume to be zero.

Nova keeps the capabilities of the invoker in a kernel-only-accessible array in the invoking thread's address space. It is located at the virtual address **OBJSP_SADDR** and stores at most **max_caps** capabilities. Line 5 extracts the capability at the index **s**$\rightarrow$ **pt()** from this array. If this capability does not refer to a portal, Nova returns prematurely with the **BAD_CAP** error code. Otherwise, Nova extracts the portal from the capability and the related receiver thread from the portal. The check in Line 14 validates that this receiver executes on the same CPU as the sender. For the information-flow analysis of this phase, three further placeholder objects are required: the portal capability, the portal and the thread control block of the receiver. There are five important points to notice:

1. Because the receiver is obtained by dereferencing two pointers (the capability target **cap.obj()** and the receiver pointer in the portal), any receiver access depends on the context in which these pointers are set (Rule [read_ptr]). The first pointer is set during the capability transfer when the invoking thread receives the portal capability. The second pointer is set at portal creation time by the creator of the portal. The analysis reveals these flows because the pointer rules update the secrecy levels of receiver accesses with the identifiers of these two pointers. An immediate consequence of these implicit flows is that neither the portal nor the worker thread of a server must be created in a context to which potential invokers of the portal capability are not cleared. A check of the create system call for portals respectively for threads reveals this leakage. The initial value of l_{ip} denotes the secrecy level of the invoker context. In a protection-parametric analysis, this secrecy level remains an abstract parameter;

2. Because the check in Line 8 returns **BAD_CAP** for non-portal capabilities, the invoking thread can probe which of its capabilities are portal capabilities. Hence, anticipating that other system calls perform similar checks, the type of a capability is revealed to all threads, which execute in the address space that holds this capability;

3. The check in Line 14 reveals whether the sender and the receiver are on the same CPU;

4. So far, no parameters are read that can be affected by the receiver. Hence, with the exception of the above information flows, no data is leaked from the receiver to the sender; and

5. So far, only the status parameter of the invoker is modified if L4-IPC terminates prematurely with **sys_finish**. Hence, the only kernel object that is modified is the sender thread.

Because the three if-statements in the Lines 8, 14 and 17 terminate prematurely, the context secrecy level l_{ip} for the subsequent Lines is at least as high as the secrecy level of the conditionals of these statements (see Section 4.5.5).

The atomic **bit_test_and_clear** operation of the second phase invalidates the last two points. In the course of its receive operation, the receiver sets the wait flag. By executing Line 17, the first sender, which finds this flag set, clears it and thereby signals to successive senders that the thread is currently unavailable. The timeout error code, which is returned if send is invoked with **Sys_ipc_send::TIMEOUT_ZERO**, returns this information. An immediate though expected consequence, is that multi-level servers have to provide at least one thread for each differently-classified client. No matter how many portals refer to a single server thread, the information flows in IPC prevent a safe processing of the requests of differently-classified clients in a single thread.

The proposed information-flow analysis is able to detect these information flows: The variable identifier **receiver→wait** appears in the secrecy level of the output parameter **status** because it is written by both IPC send and IPC receive operations.

The third phase contains two information flows from the sender to the receiver: the message and the number of untyped words that are transferred. Because message registers are located in the UTCB, the actually transferred information is the data that is located in these registers at the time of the copy. Although Line 18 denotes a word-granular copy, it is interesting to split this word-wise copy into a bit-wise copy for the purpose of the analysis. This way, only the identifier of the source message bit and identifiers from implicit information flows appear in the secrecy level of the respective receiver-side message bit. The message itself is not altered during the copy [3].

In the fourth phase, setting the continuation of the sender to **ret_user_sysexit** has no effect because the sender returns in Line 50 before the scheduler regains control. Lines 36 and 46 modify the **portal_id** field in the receiver's UTCB respectively the instruction pointer in the receiver-side register safe area. When the scheduler selects the receiver to run after the sender has enqueued this thread into the ready queue, the receiver will therefore continue at the instruction that is specified in the portal. Because both the portal id and the instruction pointer are typically configured by the receiver, no additional information is revealed in these values.

To conclude this case study, we have demonstrated a protection-parametric analysis of the send operation of the Nova IPC system call. Although the actual information-flow analysis is only described at an abstract level, it has revealed all information flows that are not encoded in the timing behavior of IPC and that are not hardware centric. The identified information flows are bidirectional and inherent for synchronous, reliable IPC designs [Sha03]. In this sense, the identified information flows are as expected.

[3] Actually, the bit-wise copy reveals only that the secrecy levels of the message are only affected by the identifiers from implicit flows. A secret stored at bit offset i is transferred to bit offset i in the message registers of the receiver.

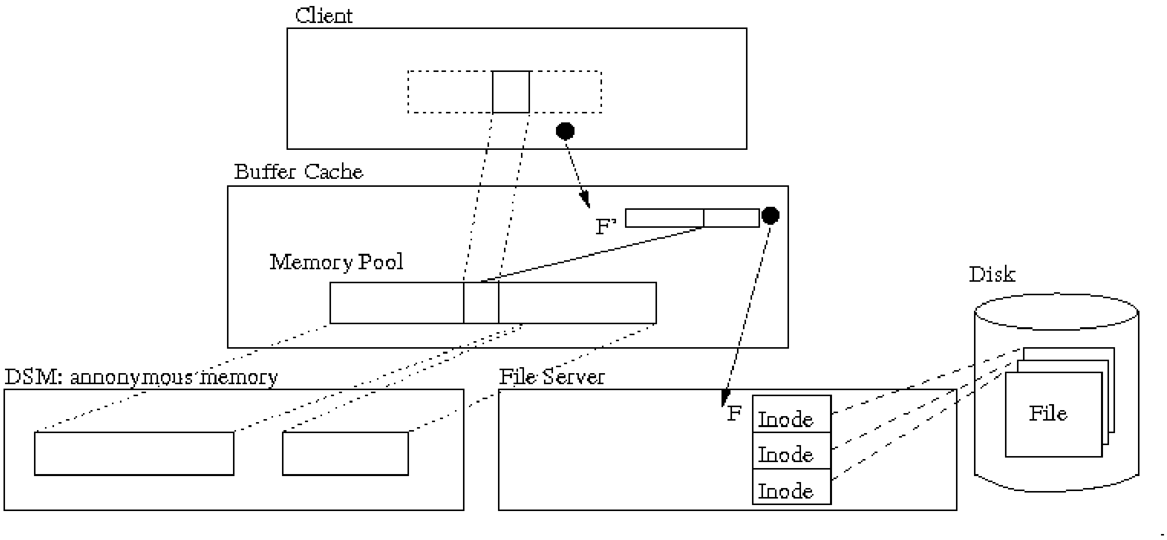

Figure 5.4.: Buffer Cache

5.3. Buffer Cache

A buffer cache is a cache, which stores the data of recently accessed file blocks (see e.g., [BC05, Chapter 15]). In L4-based systems, an implementation of a buffer cache as a buffer-cache server [GJP+00] suggests itself.

Given a memory pool, a buffer-cache server multiplexes the memory pages in this pool to cache recently accessed file data, file meta data, and meta data that the buffer cache requires for its operation. Towards their clients, buffer-cache servers typically act as dataspace managers [ADE+01]. As such, they wrap the underlying multi-level file system [4]. Let us assume that the translation of file names into file capabilities is not implemented in the buffer-cache server (this functionality can, e.g., be provided by a name or directory server). As a dataspace manager, a buffer cache typically implements interfaces that allow files to be opened and views of open files to be attached to virtual-memory regions. The buffer-cache function **get_page** is invoked by the client-side region mapper to resolve page faults in the attached view. Let us here focus on synchronous accesses to file data blocks. A file access such as **get_page** is synchronous if the accessing client thread blocks until the data is available [5]. Figure 5.4 illustrates this setup.

The buffer cache performs three operations on a memory pool:

1. When a cached file block is accessed, the buffer-cache server has to find the buffer in which this block is cached and map this buffer to the requesting client;

2. When a file block is accessed that is not cached, the buffer cache has to allocate a free buffer for this block; and

[4] Although hardware-centric covert channels are out of the scope of this thesis, I have to assume that both the underlying file system and the disk driver are free of covert channels. Otherwise, a request of a server thread for higher-classified clients could leak information through the file system or through the underlying disk [KC91].

[5] In most aspects, asynchronous accesses work similarly. The client-server communication protocols for asynchronous requests are however more complicated.

3. When no free buffers are available it has to find a block to replace.

To implement these operations, the buffer cache maintains for each buffer in the memory pool a descriptor, which denotes whether the buffer is free, when the buffer will be replaced, and, if the buffer holds file data, in which file and at which offset this data is stored. The latter is required to write back dirty buffers to the underlying file system. The data structures for finding the buffer that belongs to a certain offset of a given file are typically highly optimized and difficult to free from covert channels. For example, Linux [Tho] implements a per-file address-space object with a radix search tree to map file offsets to buffer descriptors. Similar effort is typically put into the buffer-replacement and file-system read-ahead strategies to mitigate blocking due to long disk latencies as much as possible. A modification of these strategies to eliminate covert channels in a buffer cache with a single memory pool will therefore likely come at the cost of a significant performance degradation.

One alternative solution would be to re-instantiate a buffer-cache server for each differently-classified client. However, this re-instantiation precludes a safe sharing of memory pools. In the envisaged buffer-cache server, memory pools are therefore re-instantiated but not the server itself. In Section 5.2, we have seen that the bidirectional information flows in L4-IPC prevent a safe sharing of a single thread to handle the requests of differently-classified clients. The proposed buffer-cache server therefore creates at least one thread for each such group of clients plus a set of portals whose labels refer to the file address-space objects. In addition, it maintains a set of links to the respective file address-space objects of those server threads that operate on behalf of lower-classified clients. Whenever the server thread of a higher-classified client finds that the requested file block is not in the memory pool for this client, it can therefore lookup file address-space objects of lower-classified clients to determine whether the block is already cached in their pools. If so it maps the respective block as a response to the $high$ client's **get_page** request. Allocation and replacement decisions are however limited to the own buffer cache. In particular, the buffers of write-accessible files are always allocated in the own buffer cache. Notice that write accesses to a file, which a lower-classified client can read, are already in violation of the information-flow policy.

To prove this buffer-cache server non-interference secure, we have to combine several results from the previous chapters:

Peripheral Access Control (Section 1.1)

Clients must hold only those file capabilities to which they are authorized. In particular, a $high$-classified client must not have write authority to a file that a lower classified client can read;

Secure Synchronization (Section 3.7)

File address-space objects are concurrently accessed by server threads that operate on behalf of differently classified threads. Hence, a synchronization of these accesses must not leak information about the accesses from higher classified clients to lower or incomparably classified clients. The non-interference secure locks in Section 3.7 prevent these leakages.

Information-Flow Analysis of Multi-Level Servers (Section 4.7)

The proposed information-flow analysis must check all server invocations for possible information leakages to the invoking clients or to other servers object that are used by other worker threads. In this analysis, a worker thread assumes the secrecy level of its invoking client as its context secrecy level. If, for example, such a worker thread would modify the radix search tree of a file address-space object that belongs to another worker thread, then clients could cause the worker to leak information, which the lower classified clients of the other worker thread are able to extract. The proposed information-flow analysis is able to detect these information flows.

Timing-Leak Transformations (Section 2.4.6)

With the help of the proposed information-flow analysis, only a timing-insensitive non-interference property is established. In Section 4.7.3, I therefore assume that a suitable timing-leak transformation eliminates the remaining internal timing channels. Applied to the buffer cache, this means that low-allocated buffers can be reused by higher classified clients only if the replacement of such a buffer introduces no internal timing leaks [PN92]. That is, the time required to replace a buffer must be independent of the time required to revoke access to this buffer from all higher classified clients that directly or indirectly have received a memory capability to this buffer. Unfortunately, there is no timing leak transformation that is able to establish this property for the present interface and implementations of the mapping database. Although our analysis concludes correctly that the buffer cache is timing-insensitive secure, it cannot securely be used on existing L4-family microkernels. In Section 6, I shall return to this point by sketching a modification of the mapping database, which allows for a secure implementation of the proposed buffer-cache server.

Protection-Parametric Analysis of Microkernel System Calls (Sections 4.7 and 5.2)

The bidirectional information flows of L4-IPC already prevented a single-threaded implementation of the buffer cache. However, other system calls must also be checked. In particular, as we have seen in the last paragraph, L4-unmap must be checked not to cause an un-transformable timing leak.

To conclude, the buffer cache verification succeeds to establish a timing-insensitive non-interference property for the proposed buffer-cache server. However, the subsequent timing-leak transformation fails on the mapping database. Hence, the buffer cache is not timing-sensitive non-interference secure.

5.4. AES

This section complements the three case studies with the first security-type-system-based proof of the effectiveness of Osvik's countermeasure against AES cache side-channel attacks [OST05]. For want of a type-checking tool for *Toy*, I have crafted this proof by hand. More precisely, I show that no information about the key, about the secret message or about intermediate encryption results can be leaked through the processor caches.

AES [DR99] is a round-based block cipher with 128 bit block sizes and varying key lengths. Although the AES encryption algorithm is completely defined by algebraic operations, many performance-oriented implementations use lookup tables to speed up the AES operations: SHIFTROWS, MIXCOLUMNS and SUBBYTES. In the 128 bit key version of AES, there are eight lookup tables $T_0, \dots, T_3$ and $T_0^{(10)}, \dots, T_3^{(10)}$ with 256 4-byte words each. The cipher is computed in 10 rounds. In the first nine rounds AES accesses $T_0, \dots, T_3$, in the 10^{th} round $T_0^{(10)}, \dots, T_3^{(10)}$ to accommodate for the absence of MIXCOLUMNS. Given a 16-byte

(128-bit) key $k = (k_0, \ldots, k_{15})$, the encryption algorithm starts by expanding this key to 10 round keys $K^{(r)}$, $r = 1, \ldots, 10$, which in turn are divided into four 4-byte chunks each: $K^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. For the following discussion, the construction of the above tables and the precise key expansion algorithm are of no importance [6]. Given a 16-byte plaintext $p = (p_0, \ldots, p_{15})$, each round computes an intermediate state $x^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ from the previous intermediate state. The initial state x_i^0 is computed as the **xor** of the expanded key and the plaintext as $x_i^0 = p_i \oplus k_i$, $i = 1, \ldots, 10$. Equation 5.1 defines how the intermediate states for the first 9 rounds ($r = 0, \ldots, 8$) are computed:

$$
\begin{aligned}
(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) &= T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_0^{(r+1)} \\
(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) &= T_0[x_4^{(r)}] \oplus T_1[x_9^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_1^{(r+1)} \\
(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) &= T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)} \\
(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) &= T_0[x_{12}^{(r)}] \oplus T_1[x_1^{(r)}] \oplus T_2[x_6^{(r)}] \oplus T_3[x_{11}^{(r)}] \oplus K_3^{(r+1)}
\end{aligned}
\tag{5.1}
$$

For the last round ($r = 9$), T_i is replaced by $T_i^{(10)}$. The result $x^{(10)}$ is the ciphertext.

Osvik, Shamir and Tromer [OST05] describe several software side-channel attacks based on the inspection of CPU memory cycles. They also describe corresponding countermeasures to these attacks.

An adversary may deduce the encryption key by observing which cells of the above tables the AES encryption algorithm accesses in each turn. One possibility to learn about these cells is to prepare the cache by accessing data that maps to the same sets in the cache as these tables. Because cacheline replacement occurs only between the cachelines of the same set, a table access can be detected by measuring the time required to access the preparation data. A long access time indicates a replacement of this data and hence a corresponding table access by AES.

To prevent adversaries from deducing this information, Osvik et al. propose to access the encryption tables with cacheline stride after executing the actual encryption round. This way, an adversary will always observe that the entire table was accessed.

With the help of the proposed security type system, it is possible to prove that Osvik's countermeasure avoids leakage of the key bits, of the secret message or of intermediate encryption results through the cache. To do so, we first have to describe the cacheline replacement strategy with the help of a suitable hardware side effect. Given a *Toy* implementation of this hardware side effect, the proposed security type system can then check Osvik's countermeasure together with this interleaved executing side effect for security policy violating information flows.

Adversary threads detect cache information leaks by observing the timing of preparation-data accesses. For example, a level 1 (L1) cache miss that hits in the L2 cache takes typically between 7 and 10 times longer than L1 hits. Nevertheless, a timing-insensitive analysis can reveal this leakage because the timing of memory accesses is directly correlated with the distance of the cache in which the accessed data is allocated.

To formalize the cache hardware side effect, I add for each cache i a special register R^i_{Cache}, which contains one bit for each set of this cache. Initially these bits are clear to indicate that

[6] The interested reader is referred to Daemen and Rijmen [DR99].

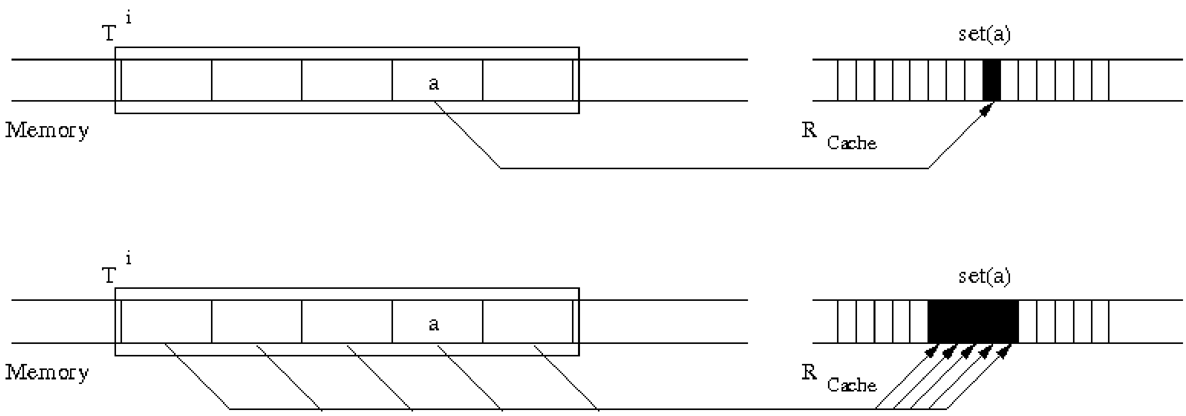

Figure 5.5.: Illustration of Osvik's countermeasure and of the hardware side effect **cse**. Black fields in R_{Cache} denote set set-bits; white fields denote clear set-bits.

the adversary has prepared the corresponding sets. If the to-be-checked program accesses a memory address a, the hardware side effect **cse(a)** sets the bit of the corresponding set in all caches where a gets allocated. A set set-bit indicates the replacement of some of the preparation data. By allowing adversary threads to read the bits in R^i_{Cache}, we have made available to these threads essentially the same information that they could have obtained from evaluating the timing of preparation-data memory accesses. For reasons of simplicity, let us here focus only on physically-tagged physically-indexed caches and on a single cache level. The proof extends straightforwardly to other cache architectures. For an n-way set associative cache, the set into which a gets stored is $set(a) := \frac{a}{n|cacheline|}\%|sets|$, where $|sets|$ is the number of sets in this cache and $|cacheline|$ is the size of a cacheline in bits.
Figure 5.5 illustrates the use of the additional registers R^i_{Cache}, the side effect **cse(a)** and the effect of the countermeasure (lower figure).
Figure 5.6 shows the C++ pseudo code p of an encryption round of AES with Osvik's countermeasure. I write **x[a .. a + 3] = y** as an abbreviation for $*$**(static_cast**$<$**long**$*>$**(&x[a])) = y**. The cacheline size is 32 bytes. The Lines 13 – 26 contain the source code of the adjusted encryption round, the Lines 28 – 31 contain the source code of Osvik's countermeasure. **cse(a)** is the cache hardware side effect. It is defined as:

```
cse(a) :=
  R_Cache[a] = 1
```

In the analysis of p, the typing rules for Lines 14 – 22 set the secrecy levels of all cacheline bits in R_{Cache} to which the table T maps to $high$.

```
1  char round;
2
3  char x[16]; // high
4  char y[16];
5  long K[10][4];
6
7  long T[4][256];
8  bool R_Cache[|sets|]; // shared with adversary threads
9
10
11 atomic { // begin non−preemptive execution
12
13   // encryption round
14   (cse(&y[0 .. 3]) ||
15    y[0  ..  3] = (cse(&T[0][x [0]])  ||  T[0][x [0]])  ^
16                  (cse(&T[1][x [5]])  ||  T[1][x [5]])  ^  ...  ^ cse(&K[round][0]) || K[round ][0]);
17   (cse(&y[4 .. 7]) ||
18    y[4  ..  7] = ...  );
19   (cse(&y[8 .. 11]) ||
20    y[8  .. 11] = ...  );
21   (cse(&y[12 .. 15]) ||
22    y[12 .. 15] = ...  );
23
24   (cse(&x[0 .. 15]) ||
25    cse(&y[0 .. 15]) ||
26    x = y );
27
28   // countermeasure
29   for (unsigned int i = 0; i < 4; i++)
30     for (unsigned int j = 0; j < 256; j+=32)
31       (cse(&T[i][ j ]) || T[i ][ j ]);
32
33 } // end non−preemptive execution
```

Figure 5.6.: AES encryption algorithm and the countermeasure against cache side-channel attacks complemented with cache-allocation information.

The reasons for this setting are twofold:

1. Because the points-to analysis does not break the cipher, it has to overestimate the elements of $T[i]$ that are accessed. Because this overestimation cannot exclude entries, all cells of the table $T[i]$ are in the set of potentially referred addresses; and

2. The table accesses of $T[i]$ and hence the accessed cachelines depend on the value of the $high$-classified key. Writes to R_{Cache} therefore depend on $high$ pointers. The targeted set-bits assume this secrecy level.

If p would allow other threads to observe the cache content at this point in time, information about the secret key could be leaked. The non-preemptive execution (Lines 11 – 33) avoids these observations.

The countermeasure in the Lines 28ff accesses T with cacheline stride. Because these stride accesses are independent of the $high$ key and because the points-to analysis is able to identify the accessed cells precisely, the secrecy levels of all set-bits of T in R_{Cache} are reset to low. A possible breach of confidentiality is avoided by removing the temporarily stored secrets before other threads can read these secrets. This concludes the prove that Osvik's countermeasure prevents leakage of the encryption table through the cache. To see that neither the key bits nor the intermediate encryption results are leaked, we have to realize that **x**, **y** and **K** are never accessed with $high$ indices. Hence, no secrets are leaked by accessing these arrays. This concludes the proof of Osvik's countermeasure.

The implementation of the hardware side effect **cse(a)** was trivial. After that, the proof follows immediately by application of the *Toy* typing rules.

6. Conclusions and Future Work

This chapter concludes my thesis and gives directions for future work. The two central contributions of this thesis are:

1. a non-interference-secure budget-enforcing fixed-priority scheduler, and
2. a sound control-flow-sensitive security type system for low-level operating-system code.

The developed solutions are a first step towards a cost-efficient provable protection of confidential data in open microkernel-based systems.

Noninterference Secure Scheduling (Section 3.3)

Two practically-feasible modifications allow a budget-enforcing fixed-priority scheduler to avoid all scheduling-related covert timing channels, even if the scheduled threads have access to precise clocks:

1. the scheduler treats possibly leaking blocked threads as if they were ready and runs higher classified ready threads to consume the budgets of these blocked threads; and
2. the scheduler defers the resumption of higher prioritized threads in situations where a lower prioritized thread could possibly leak information by executing non-preemptively.

The characterization of the budget consumption as a blocking term allows for the reuse of a large class of existing admission tests for the proposed scheduler. These tests determine whether a given set of real-time threads will meet their deadlines. Compared to the state-of-the-art schedulers for timely isolated systems — time-partitioning schedulers — the proposed non-interference secure scheduler can admit significantly more threads.

A machine-checked non-interference proof of an abstract PVS model of this scheduler substantiates that the proposed budget-enforcing fixed-priority scheduler eliminates external timing channels. The realization of this result in the theorem prover PVS proved to be valuable because a failed proof of a previous version of the scheduler revealed a flaw, which I would have likely overlooked: the two corner case situations where non-preemptively executing threads are able to leak to lower prioritized threads.

Security Type Checking Low-Level Operating-System Code (Section 4.7)

The peculiar ways in which the microkernel and the necessarily trusted multi-level servers interact with their environment pose a significant challenge to information-flow analyses for low-level operating-system code. The proposed information-flow analysis masters this challenge by first translating the to-be-checked C++ operating-system code into the non-deterministic intermediate programming language *Toy*.

The resulting *Toy* program and the interleaved executing hardware side effects are then checked with the help of a sound security type system for *Toy*. A universal lattice for shared-memory programs allows the protection-parametric analysis to cope with partially unknown or dynamic information-flow policies.

We have seen that for the special setting of low-level operating-system code, it is practically feasible and significantly more precise to repeat the analysis for all possible ways in which the control-flow non-determinism in the to-be-checked program could be resolved.

I have machine-checked the soundness proof for the security type system for the deterministic core of *Toy*.

To demonstrate the practicality of the proposed information-flow analysis, I have conducted three case studies. Chapter 5 presents the results of these case studies and the first security-type-system-based proof on the effectiveness of a countermeasure against AES cache side-channel attacks.

Future Work

This thesis opens up various directions for future work:

Tool Support: In the present work, my focus was on sound information-flow analyses for the low-level operating-system code of open microkernel-based systems. Although it is in principle well understood how these results translate into efficient type-checking tools [Mye99, Sim03, FTA02, HL09], a complete automation of the proposed protection-parametric analysis requires further work on heuristics and related static analyses. In particular, we need heuristics for the decision where to apply the standard rules for non-deterministic choice and parallel compositions and where we should check all possible resolutions of the control-flow non-determinism in these statements.

Construction Guidelines for Non-Interference-Secure Multilevel Servers: In the IPC case study in Section 5.2 and in Section 5.3, I have identified a few points where information flows in IPC affect the construction of non-interference secure multi-level servers in L4-family microkernel-based systems. A systematic investigation of these points and the development of a construction guide for provably secure multi-level servers would be interesting follow ups.

Language Support: I designed *Toy* for the specific needs of the low-level operating-system code of microkernel-based systems. For a more comprising application, additional language support is required, most notably function calls.

The two primary challenges to incorporate functions into a sound analysis of low-level operating-system code are:

1. Language support to express the conditions under which a previously checked function does not leak certain information and a corresponding mechanism to enforce that these conditions are met by the invoking code. The characterization of system-call information flows with the help of placeholder objects (see Section 4.5.4.4) is a first step in this direction; and,
2. A way to extend the results of a previously checked function to call sites where this function will be inlined.

Confinement: With the exception of Lowe et al. [ML09], confinement proofs [Boy09, EKE08, Sha00] require a-priori knowledge about all information flows that may occur in a system. On the other hand, confinement is a prerequisite to determine whether a given subsystem can obtain the authority to execute certain system calls. It is therefore interesting to combine the results of this thesis with a formal confinement result. One direction how such a combination could improve the information-flow analysis of multi-level servers would be to statically check the potential access clients may obtain if a server maps certain capabilities to these clients.

Also an integration with Lowe's work on object capability patterns [ML09] would be valuable. Lowe exploits the FDR2 model checker [For05] to automatically check non-interference properties of CSP programs that exchange capabilities. I expect that the integration of a static, security type system based analysis with a model-checking-based analysis is the key to efficiently automate protection-parametric information-flow analyses.

Noninterference-Secure Synchronization of Multi-Exemplar Resources: In Section 3.7, I have discussed the information-flow properties of single-unit resource-access protocols. Extensions to multi-unit protocols and to multi-processor resource-access protocols are therefore obvious directions for future work.

Information-Flow-Secure Capability Revocation and Secure Timeslice Donation: The current implementation of two functionalities of L4-family microkernels have to receive further attention to eliminate potentially harmful information flows: L4-unmap and the current implementation of timeslice donation [Ste04].

As we have seen in Section 5.3, the frame locks in the current implementation of L4-unmap and the inability to bound the number of directly or indirectly mapped capabilities cause information flows that preclude a safe sharing of kernel objects between differently-classified clients. These information flows can occur even if the kernel object itself can only be accessed in a read-only fashion.

Similarly, in some corner cases, the implementation of timeslice donation, which is described in [Ste04], traverses a list of threads non-preemptively that is not bounded in its size except by the number of threads in the system.

Neither of these limitations are inherent:

- For example, a mapping quota per capability limits the size of the mapping database subtree that L4-unmap has to traverse to revoke access rights. The map operation forwards the specified quota together with the capability. It thereby consumes this quota at a rate of one for each additional direct mapping and, to compensate for unmaps which include the own capability, at a rate of two for the first direct mapping;
- A node-granular lock for concurrent unmaps, which allows unmapping threads to take over the unmap lock from threads that directly or indirectly received the revoked capability from a thread in the unmapping thread's address space, avoids information flows due to lock contention; and, finally,
- Keeping blocked threads in the ready list, traversing the list of donating threads preemptively and maintaining a volatile list of current donators avoids the potentially harmful information flows in the current implementations of downward donation.

An elaborative discussion of the above points would go beyond the scope of this thesis.

A. Avoiding the Deactivation of Nonpreemptively Executing Threads

This section presents the source code of the check to avoid the abortion of non-preemptive critical sections when the execution budget of a thread depletes or when a deadline passes (see Section 3.3.5.1 on page 71). The source code of the check is the following.

```
inline
Time rdtsc() {
  Time ret;
  asm volatile ("rdtsc \n\t":"=a"(ret) :: "edx");
  return ret;
}

inline
void touch_write(void * dest) {
  asm volatile ("lock; orl \$0,%0 \n\t"::"m" (dest):);
}

inline
void touch_read(void * dest) {
  Word dummy;
  asm volatile ("movl %0,%1 \n\t" : "=R" (dummy) : "m" (dest) :);
}
    ...

class Utcb {

public:
  Time last_switch, wcet_remaining;
  Time last_release, deadline;

  volatile bool dp;
  volatile bool pending;
    ...

  inline
  bool check_remaining(Time needed) {
    Time current = rdtsc();

    return (wcet_remaining - current + last_switch > needed) &&
           (last_release + deadline - current > needed);
  }
    ...
};
```

The source code of an atomic list-enqueue operation, which relies on this check, is:

```
retry :
  utcb->dp = true;

  if ( likely (utcb->check_remaining(wcet_needed))) {

    /* touch data structures to avoid pagefaults during delayed preemption */
    touch_write(head);
    touch_write(head->_next);

    if ( unlikely (utcb->pending)) {
      /* page fault may occur during enqueue */

      // switch to kernel
      goto retry;
    }

    /* modify list */
    _next = head->_next;
    _prev = head;
    head->_next->_prev = this;
    head = this;

  } else {
    /* insufficient time to execute critical section without job-deactivating preemption

    // switch to kernel
    goto retry;
  }

  utcb->dp = false;

  if ( unlikely (utcb->pending))
    // switch to kernel
```

Bibliography

[2105] ISO/IEC Working Group JTC 1 / SC 22 / WG 21. *Commitee Draft, International Standard: Programming Languages — C*. Number [ISO/IEC 9899:TC2 - WG14/N1124]. ISO/IEC, May 2005.

[AB03] A. Aldini and M. Bernardo. Measuring the Covert Channel Bandwidth in the NRL Pump. Technical report, Universita di Urbino, 2003.

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *Static Analysis Symposium SAS'04 (also LNCS 3148)*, pages 100–115. Springer-Verlag, 2004.

[AB07] T. Amtoft and A. Banerjee. A Logic for Information Flow Analysis with an Application to Forward Slicing of Simple Imperative Programs. *Science of Computer Programming*, 64(1):3–28, 2007.

[ABB+86] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*, pages 93–113, Atlanta, GA, June 1986.

[ABR04] Gilles Arthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *Verification, Model Checking and Abstract Interpretation (VMCAI) — also LNCS 2937*. Springer-Verlag, 2004.

[Ada79] Douglas Adams. *The Hitchhiker's Guide To The Galaxy*. Completely Unexpected Productions Ltd, 1979.

[ADE+01] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill Framework for Virtual Memory Diversity. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference (ACSAC2001)*, pages 3–10, Gold Coast, Australia, January 2001.

[ADH89] V. Abrossimov, A. Demers, and C. Hauser. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP)*, pages 123–136, Lichfield Park, AZ, December 1989.

[Aga00a] J. Agat. Transforming out Timing Leaks. In *ACM Principles of Programming Languages*, Boston, Massachusetts, Jan 2000.

[Aga00b] Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. Phd thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, December 2000.

[Age72] National Security Agency. Tempest: a signal problem - the story of the discovery of various compromising radiations from communications and comsec equipment. *Cryptologic Spectrum*, 2(3), 1972.

[Age94] National Security Agency. *Specificaiton for Shielded Enclosures*, October 1994. NSA NO. 94 - 106.

[Age99] National Security Agency. *Controlled Access Protection Profile*. Fort George G. Meade, MD, USA, version 1.d edition, October 1999.

[AHS08] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-Masked Flows. In *Theoretical Computer Science*, volume 402 (2-3), pages 82–101, Essex, UK, 2008. Elsevier Science Publishers Ltd.

[AJM+06] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10 (3), August 2006.

[AR05] Gilles Arthe and Tamara Rezk. Secure information flow for a sequential java virtual machine. In *Types in Language Design and Implementation (TLDI)*, Long Beach, CA, USA, January 2005. ACM.

[ARI] ARINC. *ARINC 653-1 Standard*.

[Bak91] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *Real-Time Systems Symposium*. IEEE, 1991.

[BALL90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BC05] D. Bovet and M. Cesati. *Understanding the Linux Kernel (3rd Edition)*. O'Reilly, 2005.

[BCE+94] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. Spin – an extensible microkernel for application-specific operating system services. In *6th SIGOPS European Workshop*, pages 68–71, Schloß Dagstuhl, Germany, September 1994.

[BCG+94] P. Boucher, R. Clark, I. Greenberg, D. Jensen, and D. Wells. Towards a multilevel-secure, best-effort real-time scheduler. In *4th IFIP Working Conference on Dependable Computing for Critical Applications*, San Diego, CA, USA, Jan 1994.

[Ber04] D. J. Bernstein. Cache-Timing Attacks on AES. available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2004.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report 2547 vol. 1, MITRE, March 1973. 1996 reprint by Len LaPadula.

[BN03] A. Banerjee and D. A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *16th IEEE Computer Security Foundations Workshop*, pages 155–169. IEEE Computer Society Press, 2003.

[Boy09] A. Boyton. A verified shared capability model. In *4th Workshop on Systems Software Verification (SSV'09)*, Aachen, Germany, October 2009.

[BP03] M. Backes and B. Pfitzmann. Intransitive Non-Interference for Cryptographic Purposes. In *Symposium on Security and Privacy (SP'03)*, Oakland, California, USA, May 2003. IEEE.

[BRW06] G. Barthe, T. Rezk, and M. Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33 – 55, 2006. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005).

[BS79] M. Bishop and L. Snyder. The Transfer of Information and Authority in a Protection System. In *SOSP '79: Proceedings of the seventh ACM Symposium on Operating Systems Principles*, pages 45–54, New York, NY, USA, 1979. ACM.

[BT82] Jan A. Bergstra and J. V. Tucker. The refinement of specifications and the stabiliy of hoare's logic. In *Logic of Programs, Workshop*, pages 24–36, London, UK, 1982. Springer-Verlag.

[But05] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer Verlag, second edition edition, 2005.

[CDKM02] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. John Wiley, 2002.

[CHM02] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59, 2002.

[(Co06] Hermann Härtig (Coordinator). Open robust infrastructures, Feb 2006. http://robin.tudos.org.

[Coh78] E. S. Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, Academic Press:297–335, 1978.

[Coh96] N. H. Cohen. *Ada as a Second Language*, chapter Real-Time Systems Annex. McGraw-Hill, 1996.

[Cor09] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 25366[5-9] - 031 edition, June 2009.

[Cou96] P. Cousot. Abstract interpretation. In *Symposium on Models of Programing Languages and Computation*, volume 28 of *2*, pages 324–328. ACM Computing Surveys, 1996.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[DdEE] P. Derrin, d. Elkaduwe, and K. Elphinstone. *seL4 Reference Manual*. National ICT Australia.

[Den76] D. Denning. A lattice model of secure information flow. In *Communications of the ACM*, volume 19-5, pages 236–243, New York, NY, USA, 1976. ACM Press.

[DH66] J. B. Denssing and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *Programming Languages and Pragmatics Conference*, pages 143 – 155, San Dimas, CA, USA, March 1966. ACM.

[DL97] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[DLSU04] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 eXperimental Kernel Reference Manual, Version X.2. Technical report, University of Karlsruhe, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[dMBCS08] M. de Michiel, A. Bonenfant, H. Cass, and P. Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *14th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 161 – 166. IEEE, August 2008.

[DPHW02] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate noninterference. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[DR99] J. Daemen and V. Rijmen. *AES Proposal: Rijndael (version 2)*, September 1999.

[Dra91] Richard P. Draves. Page replacement and reference bit emulation in mach. In *Mach Symposium*. Usenix, November 1991.

[DS09] Delphine Demange and David Sands. All secrets great and small. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 207–221, Berlin, Heidelberg, 2009. Springer-Verlag.

[EES$^+$03] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 437 – 455, 2003.

[EHL98] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. L4 reference manual—MIPS R4x00, version 1.0, kernel version 70. UNSW-CSE-TR 9709, University of New South Wales, School of Computer Science, 1998.

[EKE08] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the sel4 microkernel. In *Verified Software: Theories, Tools and Experiments*, Toronto, Canada, October 2008.

[FDKHN07] J. Furuse, D. Dinh-Khac, and V. Ha-Nguyen. Flow Sensitive Information Flow Analysis for C Programs (work in progress). In *Japan-Vietnam Workshop on Software Engineering (JVSE 2007)*, Vietnum, 2007.

[FG01] R. Focardi and G. Gorrieri. Classification of security properties. Part I: Information flow. *Foundations of Security Analysis and Design*, Springer(LNCS 21711):331–396, 2001.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multi-level security of a system design. In *6th Symposium on Operating Systems Principles (SOSP)*, pages 57–65. ACM, November 1977.

[FM06] N. De Francesco and L. Martini. Abstract Interpretation to Check Secure Information Flow in programs with input-output security annotations. *Lecture Notes in Computer Science (LNCS)*, 3866:63–80, February 2006.

[FN79] R. J. Feiertag and P. G. Neumann. The Foundations of a Provably Secure Operating System (PSOS). In *National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[For05] Formal Systems (Europe), Limited. *FDR2 User Manual*, 2005.

[Fou91] Free Software Foundation. Gnu general public license version 2. http://www.gnu.org/licenses/gpl-2.0.html, June 1991.

[Fra83] L. J. Fraim. Scomp: A solution to the multilevel security problem. *Computer*, 16(7):26–34, 1983.

[FS96] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, Seattle, WA, USA, October 1996.

[FTA02] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, Berlin, Germany, June 2002. ACM.

[Gal93] P. Gallagher. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, 1993.

[GHRS05] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, 2005.

[GJ77] M. R. Garey and D.S. Johnson. Two processor scheduling with start time and deadlines. *SIAM Journal of Computing*, 6, 1977.

[GJP+00] A. Geffault, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In *9th SIGOPS European Workshop*, pages 109 – 114, Koldingfjord, Kolding, Denmark, September 2000. ACM.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, 1982.

[GMF79] M. Gasser, J. K. Millen, and W. F.Wilson. A note on information flow into arrays. Technical report m79-234, MITRE Corporation, Bedford, MA, USA, December 1979.

[Gra93] James W. Gray, III. On Introducing Noise into the Bus-Contention Channel. In *IEEE Symposium on Security and Privacy*, page 90, Washington, DC, USA, 1993. IEEE Computer Society.

[Hae03] Andreas Haeberlen. Managing kernel memory resources from user level. Diploma thesis, Universität Karlsruhe, Karlsruhe, Germany, April 2003. Supervisor: Dr. K. Elphinstone, Dr. V. Uhlig, Prof. Dr. A. Schmitt.

[Han99] Steven M. Hand. Self-paging in the nemesis operating system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[Har85] N. Hardy. The KeyKOS Architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HERH93] G. Heiser, K. Elphinstone, S. Russell, and G. R. Hellestrand. A distributed single address-space operating system supporting persistence. SCS&E Report 9302, Univ. of New South Wales, School of Computer Science, Kensington, Australia, March 1993.

[HHF+05] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *First International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, California, USA, Dec. 2005.

[Hil92] D. Hildebrand. An architectural overview of QNX. In *1st USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[HK93] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Summer USENIX Conference*, pages 147–160, Cincinnati, OH, June 1993.

[HKMY87] T. J. Haigh, R. A. Kemmerer, J. McHugh, and W. D. Young. An Experience Using Two Covert Channel Analysis Techniques on a Real System Design. *IEEE Transactions on Software Engineering*, 13(2):157–168, 1987.

[HL09] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Principles of Programming Languages (POPL'09)*, Savannah, Georgia, USA, January 2009. ACM.

[HLR+01] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 (8):666 – 677, August 1978.

[Hoh96] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, August 1996. In German; with English slides. Available from URL: `http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/`.

[Hoh02] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[HPS01] M. Heisel, A. Pfitzmann, and T. Santen. Confidentiality-Preserving Refinement. In *CSFW '01: Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 295 ff., Washington, DC, USA, 2001. IEEE Computer Society.

[HS05] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005.

[HS06] S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006. ACM.

[Hu91] W. Hu. Reducing timing channels with fuzzy time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, USA, May 1991.

[Hu92] W. Hu. Lattice Scheduling and Covert Channels. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1992.

[HWF05] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.

[HWS03] C. Helmuth, A. Westfeld, and M. Sobirey. μSINA - Eine mikrokern-basierte Systemarchitektur für sichere Systemkomponenten. In *Deutscher IT-Sicherheitskongress des BSI*, volume 8 of *IT-Sicherheit im verteilten Chaos*, pages 439–453. Secumedia-Verlag Ingelsheim, May 2003.

[IEE93] IEEE. *Portable Operating System Interface (POSIX) 1b - Real-Time Extensions*, std 1003.1b edition, 1993.

[II09] ECRYPT II. Yearly report on algorithms and keysizes (2008-2009). Technical Report D.SPA.7 Rev 1.0, ICT-2007-216676, July 2009.

[IK08] Futoshi Iwama and Naoki Kobayashi. A new type system for jvm lock primitives. *New Generation Computing*, 26(2):125–170, Feb. 2008.

[Inc95] Gemini Computers Inc. Gemsos evaluation report. Technical report, National Computer Security Center, 1995.

[Inc09] Green Hills Software Inc. Integrity real-time operating system. available at http://www.ghs.com/products/rtos/integrity.html, 2009.

[Int06] Intel Corp. *LaGrande Technology - Preliminary Architecture Specification*, March 2006. DMA protection is superseded by VT-d.

[Jac89] J. Jacob. On the derivation of secure components. In *IEEE Symposium on Security and Privacy*, pages 242–247, May 1989.

[Jen92] E.D. Jensen. Asynchronous decentralized realtime computer systems. In *NATO Advanced Study Institute on Real-Time Computing*, Saint Martin, October 1992.

[JPW05] B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *WITS '05: Proceedings of the 2005 workshop on Issues in the theory of security*, pages 50–56, New York, NY, USA, 2005. ACM Press.

[Kar88] P. A. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, Wolfson College, University of Cambridge, March 1988.

[KC91] P. A. Karger and Wray J. C. Storage channels in disk arm optimization. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61, May 1991.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heise, June Andronick, David Cock, Phillip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafael Klanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verifiation of an os kernel. In *22nd Symposium on Operating Systems Principles*, Big Sky, Montana, USA, October 2009. ACM.

[Kem83] Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*, 1(3):256–277, 1983.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology - Crypto99 (also LNCS 1666)*. Springer Verlag, 1999.

[KM07] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal on Information Security*, 6(2-3):107–131, 2007.

[Kop98] H. Kopetz. The time-triggered architecture. In *ISORC*, 1998.

[KP91] R. A. Kemmerer and P. A. Porras. Covert Flow Trees: A Visual Approach to Analyzing Covert Storage Channels. *IEEE Transactions on Software Engineering*, 17(11):1166–1185, 1991.

[Kru02] S. D. Krueger. System protection map. Us patent 6775750, Texas Instruments Inc., October 2002.

[KS02] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the multics security evaluation. *Computer Security Applications Conference, Annual*, 0:119, 2002.

[KT96] R. A. Kemmerer and T. Taylor. A Modular Covert Channel Analysis Methodology for Trusted DG/UX. In *ACSAC '96: Proceedings of the 12th Annual Computer Security Applications Conference*, page 224, Washington, DC, USA, 1996. IEEE Computer Society.

[KV05] B. Kauer and M. Völp. *L4.Sec Preliminary Microkernel Reference Manual*. Technische Universität Dresden, Oct 2005. Available at: http://os.inf.tu-dresden.de/L4/L4.Sec.

[KWS97] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler Conscious Synchronization. *ACM Transactions on Computer Systems*, Feb. 1997.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Frans M. Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, Stevenson, Washington, USA, 2007. ACM.

[KZB+91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[Lac04] Adam Lackorzynski. L4linux porting optimizations. Master's thesis, TU-Dresden, Dresden, Germany, April 2004.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[LEA07] R. Leslie, L. Erkok, and F. Andersen. Formalizing information flow in a haskell hypervisor. In *First International Workshop on Microkernels for Embedded Systems (MIKES 2007)*, January 2007.

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Chatham (Cape Cod), MA, May 1997.

[Les06] R. Leslie. Dynamic Intransitive Noninterference. In *International Symposium on Secure Software Engineering (ISSSE 2006)*, McLean, VA, USA, March 2006. IEEE.

[LHH97] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, March 1992. Springer.

[Lie93] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[Lie95] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Lie96] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[Lie99] J. Liedtke. L4 nucleus version x reference manual (x86). Technical report, September 1999.

[Lip75] R. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), Dec 1975.

[Liu00] J. W. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in an Hard-Real-Time Environment. *Journal of the ACM*, 20.1:46–61, Jan 1973.

[Low02] Gavin Lowe. Quantifying information flow. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 18, Washington, DC, USA, 2002. IEEE Computer Society.

[Low04] G. Lowe. Semantic models of information flow. *Theoretical Computer Science*, 315:209 – 256, 2004.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[Ltd] ARM Ltd. http://www.arm.com.

[LU02] J. Loughry and D. A. Umphress. Information leakage from optical emanations. *ACM Transactions on Information and System Security*, 5(3):262–289, August 2002.

[LUV05] Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to dataflow analyses of imperative languages. In *2rd Workshop on Applied Semantics (APPSEM'05)*, September 2005.

[LUY+08] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Soft layering for virtual machines. In *Proceedings of the 13th IEEE Asia-Pacific Computer Systems Architecture Conference*, Hsinchu, Taiwan, August 4–6 2008. Best Paper Award.

[LW09] Adam Lackorzynski and Alexander Warg. Taming Subsystems - Capabilities as Universal Resource Access Control in L4. In *IIES'09: Second Workshop on Isolation and Integration in Embedded Systems (Eurosys 2009 affiliated Workshop)*, March 2009.

[Lyn] Lynuxworks. Lynxos: Partitioning operating systems vs. process-based operating systems. available at http://www.lynuxworks.com/products/whitepapers/partition.php.

[LZ06] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, page 16, Washington, DC, USA, 2006. IEEE Computer Society.

[MAWF98] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In *7th International Conference on Compiler Construction (LNCS 1383)*. Springer Verlag, 1998.

[MB05] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[McC90] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, pages 563 – 568, 1990.

[MDP96] D. Mosberger, P. Druschel, and L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software — Practive and Experience*, 26(1):1–23, 1996.

[Meh05] Frank Mehnert. *Kapselung von Standard-Betriebssystemen*. PhD thesis, Technische Universität Dresden, July 2005. in German.

[Mil89a] Jonathan K. Millen. Finite-state noiseless covert channels. In *IEEE Computer Security Foundations Workshop*, pages 81–86, Kenmare, County Kerry, Ireland, March 1989.

[Mil89b] R. Millner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] Jonathan Millen. 20 years of covert channel modeling and analysis. *Security and Privacy, IEEE Symposium on*, 0:0113, 1999.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142. ACM, 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Security and Privacy*, Oakland, California, May 1998.

[ML09] T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. In *6th Workshop on Formal Aspects of Security and Trust (FAST '09)*, 2009.

[MP85] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Symposium on Principles of Programming Languages*, pages 37 – 51, New Orleans, ,USA, January 2985. ACM.

[MSZ06] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *IEEE Journal on Computer Security*, pages 157 – 196, 2006.

[Mye99] A. C. Myers. Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228 – 241, San Antonio, Texas, USA, January 1999. ACM.

[Noh08] Karsten Nohl. Cryptanalysis of crypto-1. Technical report, University of Virginia, march 2008. available at http://www.cs.virginia.edu/˜kn5f/pdf/Mifare.Cryptanalysis.pdf.

[Nor98] Michael Norrish. C formalized in HOL. Technical Report UCAM-CL-TR-452, University of Cambridge, Cambridge, UK, December 1998.

[Nor99] Michael Norrish. Deterministic expressions in c. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 147–161, London, UK, 1999. Springer-Verlag.

[OCsC06] K. O'Neill, M. Clarkson, and s. Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[Ohe04] D. Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In *Computer Security – ESORICS 2004*, 2004.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[OST05] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptology ePrint Archive, Report 2005/271*, 2005.

[otAH88] Department of the Army Headquaters. *AR 380-5 Department of the Army Information Security Program*. US Army, Washington, DC, USA, March 1988.

[Pap98] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Athens, Greece, February 1998.

[PC09] ANSI Core Language Working Group (INCITS PL22.16) and ISO (WG21) C++ Standard Committee. *Working Draft, Standard for Programming Language C++*, volume [N2914=09-0104]. ISO/IEC, June 2009.

[Pet09] Michael Peter. L4 cx. personal communication, December 2009.

[PN92] N. E. Proctor and P. G. Neumann. Architectural Implications of Covert Channels. In *15th National Computer Security Conference*, pages 28–43, Baltimore, Maryland, USA, October 1992.

[PS03] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Transactions on Programming Languages and Systems (TOPLAS)*, volume 25 of *1*, pages 117–158. ACM, January 2003.

[PSLW09] M. Peter, H. Schild, A. Lackorzynski, and A. Warg. Virtual machines jailed - virtualization in systems with small trusted computing bases. In *VTDS'09 Workshop on Virtualization Technology for Dependable Systems*, March 2009.

[RD82] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[Reu03] Lars Reuther. *L4 Region Mapper Reference Manual*. Technische Universität Dresden, Dresden, Germany, 2003. available at:
http://os.inf.tu-dresden.de/l4env/doc/html/l4rm/.

[RK79] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communinactions of the ACM*, 22(2):115–123, 1979.

[RS01] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, Washington, DC, USA, 2001. IEEE Computer Society.

[RS06] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *19th IEEE Computer Security Foundations Workshop*, Venice, Italy, July 2006.

[RS09] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *Journal of Logic and Algebraic Programming*, 78(7):593–618, November 2009.

[Rus92] J. Rushby. Noninterference, Transitivity, and Channel-control Security Policies. Technical Report CSL-92-2, Stanford Research Institute, 1992.

[Rya90] P. Ryan. A CSP formulation of non-interference and unwinding. In *Computer Security Foundation Workshop*. IEEE, 1990.

[Ryd03] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Lecture Notes on Computer Science (LNCS 2622)*, pages 126–137. Springer Verlag, 2003.

[SA98] Raymie Stata and Martin Abadi. A type system for java bytecode subroutines. In *Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, USA, January 1998. ACM.

[SA07] J. S. Shapiro and J. W. Adams. *Coyotos Microkernel Specification (V 0.6+)*. The EROS Group LLC, Sept. 2007. avilable at
http://www.coyotos.org/docs/ukernel/spec.html.

[Sab01a] A. Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *4th International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, July 2001. Springer-Verlag.

[Sab01b] Andrei Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs*. PhD thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, May 2001.

[SAF06] J. Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *7th Annual IEEE Information Assurance Workshop*, West Point, NY, USA, June 2006.

[SC08a] Richard M. Stallman and GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, gcc version 4.5.0 edition, 2008.

[SC08b] Richard M. Stallman and GCC Developer Community. *Using the GNU Compiler Collection*. Free Software Foundation, gcc version 4.5.0 (pre-release) edition, 2008.

[Sch96] S. Schönberg. L4 on Alpha, design and implementation. Technical Report CS-TR-407, University of Cambridge, 1996.

[SCS77] Michael D. Schroeder, David D. Clark, and Jerome H. Saltzer. The Multics Kernel Design Project. In *6th ACM Symposium on Operating Systems Principles (SOSP)*, West Lafayette, Indiana, USA, November 1977. ACM.

[SESS96] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, October 1996.

[SGLS77] M. Schaeffer, B. Gold, R. Linde, and J. Scheid. Program Confinement in KVM/370. In *National ACM Conference*, Oct. 1977.

[Sha99] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, April 1999.

[Sha00] J.S. Shapiro. Verifying the EROS Confinement Mechanism. In *IEEE Symposium on Security and Privacy*, 2000.

[Sha03] Jonathan S. Shapiro. Vulnerabilities in synchronous ipc designs. In *Symposium on Security and Privacy*, Oakland, CA, USA, 2003. IEEE.

[Sha06] Jonathan Shapiro. Programming language challenges in systems codes: why systems programmers still use c, and what to do about it. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 9, New York, NY, USA, 2006. ACM.

[Sim03] V. Simonet. *Flow Caml*. Instinut National de Recherche en Informatique et en Automatique, July 2003. available at
http://www.normalesup.org/˜simonet/soft/flowcaml/manual/index.html.

[SK08] U. Steinberg and B. Kauer. Nova os virtualization architecture. OSDI - Poster Session, December 2008.

[SM02] Andrei Sabelfeld and Heiko Mantel. Securing communication in a concurrent language. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 376–394, London, UK, 2002. Springer-Verlag.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, January 2003.

[Smi01] Richard E. Smith. Cost Profile of a Highly Assured, Secure Operating System. *ACM Transactions on Information and System Security*, 4(1):72–101, 2001.

[SPHH06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. *SIGOPS Operating Systems Review*, 40(4):161–174, 2006.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. *IEEE Transaction on Computers*, 39, 1990.

[SRS+02] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verified formal security models for multiapplicative smart cards. *Journal of Computer Security*, 10(4):339–367, 2002.

[SS99] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *8th European Symposium on Programming (also LNCS 1576)*, pages 40 – 58, Amsterdam, The Netherlands, March 1999. Springer Verlag.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th Computer Security Foundations Workshop*, pages 255 – 269. IEEE, June 2005.

[Ste04] U. Steinberg. Quality-assuring scheduling in the fiasco microkernel. Diploma thesis, Technische Universität Dresden, Dresden, Germany, March 2004. Supervisor: Dr.-Ing. Michael Hohmuth, Jean Wolter.

[Ste09a] Udo Steinberg. *NOVA Microhypervisor Interface Specification*. Technische Universität Dresden, Dresden, Germany, December 2009. available at http://hypervisor.org.

[Ste09b] Udo Steinberg. NOVA Microhypervisor prerelease version 0.1. available at http://hypervisor.org, 2009.

[Sti00] M. Stiegler. *The E Language in a Walnut*, chapter 4.4. Capability Patterns - Powerbox Capability Manager. M. Stiegler, 2000. available at http://www.skyhunter.com/marcs/ewalnut.html.

[Sto07] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM SIGOPS Operating System Review - Special Topics on Secure Small-Kernel Systems*, July 2007.

[Str03] Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[SV98] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *25th Symposium on Principles of Programming Languages (POPL)*, San Diego, California, USA, January 1998. ACM.

[SVJ+05] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. Linwood Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. IBM Research Report RC23511, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, USA, February 2005.

[SWH05] U. Steinberg, J. Wolter, and H. Härtig. Fast Component Interaction for Real-Time Systems. In *17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[SWM00] F. Smith, D. Walker, and G. Morisett. Alias types. In *European Symposium on Programming*, March 2000.

[TD08] S. Tlili and M. Debbabi. Type and Effect Annotrations for Safe Memory Access in C. In *Third International Conference on Availability, Reliability and Security (ARES)*, pages 302–309, Barcelona, Spain, April 2008. IEEE.

[TGC87] C. R. Tsai, V. D. Gligor, and C. S. Chandersekaran. A formal method for the identification of covert storage channels in source code. In *IEEE Symposium on Security and Privacy*, pages 74–86, Oakland, CA, USA, 1987.

[Tho] Linus Thorvals. Linux kernel 2.6.

[Tro93] J. T. Trostle. Modelling a fuzzy time system. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 82–89, Oakland, CA, USA, May 1993.

[TVW09] Hendrik Tews, Marcus Völp, and Tjark Weber. Formal memory models for the verification of low-level operating-system code. *Journal of Automated Reasoning - Special Issue on Operating System Verification*, 42(2):189 – 227, April 2009.

[TWM+09] Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 109–120, Washington, DC, USA, March 2009.

[TWV+08] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro-hypervisor verification. Robin deliverable d.13, Radboud Universiteit Nijmegen, Nijmegen, The Netherlands, April 2008.

[VEK+07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[VHH08a] M. Völp, C. J. Hamann, and H. Härtig. Avoiding Timing Channels in Fixed-Priority Schedulers - PVS Sources. available at http://os.inf.tu-dresden.de/˜voelp/sources/sec_rt_trans.tgz, 2008.

[VHH08b] Marcus Völp, Claude J. Hamann, and Hermann Härtig. Avoiding timing channels in fixed priority schedulers. In *Asian Conference on Computer and Communication Security (ASIACCS '08)*, Tokyo, Japan, March 2008. ACM.

[vOWL03] David von Oheimb, Georg Walter, and Volkmar Lotz. A formal security model of the infineon sle 88 smart card memory managment. In *8th European Symposium on Research in Computer Security (ESORICS)*, pages 217–234, October 2003.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, December 1996.

[Völ08a] Marcus Völp. Statically checking confidentiality of shared-memory programs with dynamic labels. In *3rd International Conference on Availability, Reliability and Security (ARES)*, Barcelona, Spain, March 2008. IEEE.

[Völ08b] Marcus Völp. Statically Checking Confidentiality of Shared-Memory Programs with Dynamic Labels - PVS Sources. available at http://os.inf.tu-dresden.de/˜voelp/sources/dyn_sm.tgz, 2008.

[Völ10] Marcus Völp. PhD thesis - PVS sources. available at http://os.inf.tu-dresden.de/˜voelp/sources/thesis/index.html, 2010.

[Wal93] Charles Wallace. The semantics of the c++ programming language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1993.

[Wal95] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, 1974.

[WH08] C. Weinhold and H. Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *EuroSys*, Glasgow, Scotland, April 2008.

[Wik] Wikipedia. Set theory. available at http://en.wikipedia.org/wiki/Set_theory.

[Win93] Glynn Winskel. *The formal Semantics of Programming Languages — An Introduction*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, USA, 1993.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1 – 12. ACM, June 1995.

[WL02] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS 2002 (also Lecture Notes on Computer Science 2477)*, pages 180–195. Springer Verlag, 2002.

[WL07] Z. Wand and R. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA)*, pages 495 – 505, San Diego, CA, USA, 2007. ACM.

[WL10] Alexander Warg and Adam Lackorzynski. The fiasco.oc kernel. avail. at http://os.inf.tu-dresden.de/fiasco, June 2010.

[Wu] Qiang Wu. *Survey of Alias Analysis*. Princeton University, Princeton, NJ, USA. available at http://www.cs.princeton.edu/˜jqwu/Memory.

[WW94] C. A: Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, USA, November 1994.

[WW95] C. A: Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1995.

[Zan02] M. Zanotti. Security Typings by Abstract Interpretation. *Lecture Notes in Computer Science (LNCS) — also SAS'02*, 2477:360–375, August 2002.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[ZPS99] K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, pages 277–291, Kiawah Island, SC, December 1999.

Index

absolute deadline, 50
abstract data type, 42
abstract interpretation, 29
access-control mechanism, 3, 8, 179
  capability, 3
access-control policy, 18
accounting, 74
admission test, 7, 45, 99
AES, 180
aperiodic, 53
approximate non-interference, 11
assembly-level security type system, 27

bandwidth, 11
bandwidth-preserving server, 53
basic priority ceiling protocol, 109
budget consumer, 74
budget-enforcing scheduler, 6, 35, 45, 51
buffer cache, 178

capability, 3
classification, 19
clearance, 4, 19
compartment, 39
compiler memory barrier, 118, 127
compiler optimization, 5
completeness, 5
computational non-interference, 24
confinement, 39, 131
context, 4
control-flow sensitive security type system, 28, 156
control-flow-insensitive security type system, 27
control-flow-sensitive security type system, 8, 117
countermeasure, 60
Countermeasure I, 61
covert channel, 1, 7, 15
  bandwidth, 11
  external timing channel, 15
  hardware-centric, 13, 16
  internal timing channel, 16
  noise, 16
  scheduler channel, 7, 17, 40, 45, 57
  software-centric, 16
  storage channel, 15
  timing channel, 5, 15
critical instant, 47, 98
cross copying, 32

data space, 37
data type, 137
declassification, 20
direct influence, 57, 77
domain, 18
dominates relation, 18
donation ceiling, 110
dynamic information-flow controls, 12
dynamic information-flow policy, 20

effective release time, 103
end-of-release preemption, 71
entity, 19
explicit information flow, 17
extended real, 137
external timing channel, 15
external timing leak, 17

field-sensitive security type system, 125
FIFO, 76
fish, v
fixed point, 29
fixed-priority scheduler, 45
fully interruptible, 60, 73
fuzzy time, 7

hardware side effect, 121
hardware-centric covert channel, 13, 16
hierarchical scheduling, 104

idle thread, 61
implicit information flow, 17
incomparable, 19
indirect influence, 58, 77
information flow
  explicit, 17
  external timing leak, 17
  implicit, 17
  internal timing leak, 17, 78
  probabilistic, 18
  refinement, 18
  termination leak, 17
information-flow policy, 18
  dynamic, 20
  intransitive, 19, 65
  lattice model, 4, 18
  partially-unknown, 30, 128
  transitive, 64
input oracle, 146
inter-process communication, 3, 34
intermediate programming language, 9
  Toy, 9, 134
internal timing channel, 16
internal timing leak, 17, 78
interpreted data type, 137
intransitive information-flow policy, 19, 65
intransitive pass, 19, 46
  intransitive point, 19, 66
intransitive point, 19, 66
isolation
  spatial, 3, 6
  temporal, 3, 6, 98

join point, 30

L4, 12, 34, 172
lattice, 4, 19
  universal, 31, 39, 128
lattice model, 4, 18
lattice scheduler, 40, 75
leakage, 1
learned secrets, 129, 152
loader, 36
local respect, 26
lock, 144
lock similar, 134, 146
locking discipline, 144
loop-bound analysis, 34
low-level language feature, 124

maximum preemption delay, 69
memory management unit, 121
memory model, 131, 135
microhypervisor, 3
microkernel, 2
  L4, 12, 34, 172
multilevel component, 2
multilevel server, 2

noise, 16
non-allocated temporary, 136
non-disclosure, 20
non-influence, 21, 165
non-interference, 11, 21, 93, 165
  approximate, 11
  computational, 24
  non-influence, 21, 165
  non-leakage, 24
  observationally indistinguishable, 21, 163
  quantitative, 11
  unwinding, 8, 23
non-leakage, 24
non-preemptive execution, 59, 77, 102
non-volatile memory access, 127
not-a-thing
  quiet, 137
  signalled, 137

observationally indistinguishable, 21, 163
occlusion, 25
open system, 3

pager, 36
partially-unknown information-flow policy, 30, 128
points-to analyses, 33
points-to analysis, 126
potential access, 144
precedence constraint, 102
premature termination, 136, 150
priority-inheritance protocol, 108
probabilistic information flow, 18
prohibition time, 99
proportional-share scheduler, 55, 78
protection-parametric analysis, 129, 180
PVS, 13, 41

quantitative non-interference, 11
quiet not-a-thing, 137

rate monotonic scheduling, 57, 98
real time, 98
refinement, 8
refinement information flow, 18
region mapper, 36, 178
resource access, 107
  basic priority ceiling protocol, 109
  donation ceiling, 110
  priority-inheritance protocol, 108
  stack-based priority-ceiling protocol, 109
resource contention, 114
response time, 47
ReThMo, 46
Round Robin, 76

scheduler
  admission test, 7, 45, 99
  budget-enforcing, 6, 35, 45, 51
  fixed-priority, 45
  proportional-share, 55, 78
  timeslice donation, 36, 106
scheduler channel, 7, 17, 40, 45, 57
scheduling
  rate monotonic, 57, 98
secrecy level
  incomparable, 19
security policy
  access-control policy, 18
  information-flow policy, 18
security type system, 4, 26
  assembly-level, 27
  control-flow sensitive, 28, 156
  control-flow-insensitive, 27
  control-flow-sensitive, 8, 117
  field-sensitive, 125
  timing-insensitive, 5
  tool, 11, 33
  typing judgement, 27
self suspension, 100
semantics, 135
  data type, 137
  memory model, 131, 135
  small step, 138, 140
signalled not-a-thing, 137
singleton set, 33
small step semantics, 138, 140
software-centric covert channel, 16
soundness, 5, 163
spatial isolation, 3, 6
sporadic, 53
stack-based priority-ceiling protocol, 109
state transformer, 85
step consistency, 26
storage channel, 15
strictly periodic, 52
strong update, 33
structural induction, 42
support, 126

task model
  aperiodic, 53
  ReThMo, 46
  sporadic, 53
  strictly periodic, 52
temporal isolation, 3, 6, 98
termination leak, 17
theorem prover
  PVS, 13, 41
time demand, 98
time-partitioned system, 7, 54
timeslice donation, 36, 106
timing channel, 5, 15
timing-insensitive security type system, 5
timing-leak transformation, 5, 11, 16, 32, 180
  cross copying, 32
  transactional branching, 32
  unification, 32
tool, 11, 33
Toy, 9, 134
transactional branching, 32
transitive information-flow policy, 64
translation-lookaside buffer, 121
trusted computing base, 3
type correctness constraint, 43
typing judgement, 27
typing rule, 133, 158

unification, 32
universal lattice, 31, 39, 128
unsafe typing discipline, 125
unwinding, 8, 23
utilization, 100

variable clearance, 4
virtual-memory alias, 126

weak update, 33, 153