Provable Protection of Confidential Data in Microkernel-Based Systems

Diplom Informatiker
Marcus Rolf Völp

Summary of my Doctoral Thesis

submitted to the

Faculty for Computer Science
Technische Universität Dresden

in partial fulfillment of the requirements for the degree of:

Doktoringenieur (Dr. Ing.)

submitted:	August 10, 2010
date of defence:	January 31, 2011

Thesis Committee:

Prof. Dr. rer. nat. Hermann Härtig (Supervisor)	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	Raboud Universiteit Nijmegen

Provable Protection of Confidential Data in Microkernel-Based Systems

Modern computer systems process increasing amounts of private, sensitive, and valuable information. Yet, despite significant academic and industrial efforts, most of today's operating systems (OSs) do not protect confidential data against unauthorized disclosure over covert channels. Large trusted computing bases and high covert-channel analysis costs are two factors contributing to this situation.

In my thesis "*Provable Protection of Confidential Data in Microkernel-Based Systems*" [VÏ0], I approach the visionary goal of a cost-efficient, provable, and perfect protection of confidential data against leakage over covert communication channels. To achieve this goal, I combine the complementary strength of microkernels and of security type systems. This document summarizes the major challenges and results of my thesis.

1 Introduction

Microkernel-based systems significantly reduce the per-application trusted computing base, and hence, the amount of code that must be relied upon to protect the confidentiality of the data they process. Open microkernel-based systems are a particularly interesting class of microkernel-based systems:

- they co-host real-time-critical [DL97] and security-sensitive [Här02] applications next to legacy OSs and their applications; and
- they facilitate a construction principle to split complex security-sensitive and real-time-critical applications into a critical core and into non-critical parts, which reuse the code and functionality of not necessarily trustworthy legacy OSs [HPHS04].

Such a split is only possible because the multilevel components of the open system isolate parts with different criticality in both a spacial and in a temporal manner. In a microkernel-based system, these *multilevel components* are the microkernel and those necessarily trusted low-level operating-system servers, which operate on behalf of differently classified clients and which cannot be re-instantiated for each such client.

What justifies our trust in multilevel components to protect the confidentiality of sensitive data? Ideally, each such component comes with a formal proof, which confirms its isolation capabilities. However, the costs of traditional formal and semi-formal methods to establish these results are significant, both in terms of highly-skilled personnel and in terms of labor hours [Smi01, HKMY87]. This and the microkernel's inability to constrain application-internal leakage are what brings security type systems into play.

Sound security type systems [VSI96] and related language-based approaches to information-flow security [SM03] are powerful source-level analyses to prove the absence of security policy violating information flows and hence the protection of confidential data in application-level programs. However, the peculiarities of low-level operating-system code prevents an immediate application of traditional security type systems to prove data confidentiality in open microkernel-based systems.

The here summarized thesis addresses these challenges as follows:

1. a scheduler for the microkernel is developed and proven non-interference secure. It avoids leakage over scheduling-related (i.e., external) covert timing channels;
2. a security type system for the low-level operating-system code of open microkernel-based systems is developed and proven sound to avoid leakage over software-centric covert storage channels; and finally

3. existing timing-leak transformations [Aga00] are used to eliminate the remaining internal timing channels.

The remainder of this summary is structured as follows: Section 2 highlights the possibilities to leak information through fixed-priority schedulers and presents a non-interference-secure budget-enforcing fixed-priority scheduler, which avoids these illegal information flows in open microkernel-based systems. Section 3.3 introduces the peculiarities of low-level operating-system code and how the proposed analysis checks low-level operating-system code for information leakage over storage channels. Section 4 summarizes the existing timing-leak transformations before it focuses on secure resource usage in real-time systems. Section 5 summarizes the case studies I have performed. Section 6 concludes this summary.

2 Avoiding Timing Leaks in Fixed-Priority Schedules

In fixed-priority schedulers, altering a thread's execution or blocking behavior in a secret-dependent way constitutes an information flow to lower or equally prioritized threads. If such a leaking thread blocks, an unmodified fixed-priority scheduler will select a lower or equally prioritized thread to run; if it runs, the selection will be deferred to a later point in time.

In addition to that, non-preemptively executing threads can defer the points in time when higher or equally prioritized threads resume their execution after they are released or after they have been blocked. Let us call these possibilities for information leakage *direct influence* respectively *influence due to non-preemptive execution*.

2.1 An Information-Flow-Secure Budget-Enforcing Fixed-Priority Scheduler

To avoid information leakage due to direct influences, due to non-preemptive execution, and the indirect leakages, which occur when thread manipulate the timing of legitimate messages from the influenced threads, I introduce two practically feasible modifications to a budget-enforcing fixed-priority scheduler:

- *Countermeasure I:* to treat possibly leaking blocked threads as if they were ready; and
- *Countermeasure II:* to defer the points in time when higher prioritized threads resume their execution after they are released or after they have blocked.

Because a budget-enforcing scheduler deactivates a job of a thread after it has executed and blocked longer than its combined execution and blocking budget, treating possibly leaking blocked jobs as ready defers the execution of lower or equally prioritized threads to the point in time when such a job has exhausted its budget or when its deadline has passed. Direct influence and hence also an indirect influence through message timing is avoided. To ensure that a blocked thread consumes its budget, the scheduler runs *budget-consumer threads* instead of the blocked thread. Budget consumers are selected according to ascending secrecy levels to avoid leakage to the blocked thread and to previously chosen blocked budget consumers. Following Hu [Hu92], I call the resulting scheduler the *budget-enforcing fixed-priority lattice scheduler*.

Both countermeasures are controlled by static (i.e., off-line computable) predicates, which determine when and for which threads the respective countermeasure should be applied. These predicates are:

$$p_{transitive}(\tau_h, t) := \exists\, \tau_l \in T.\ prio(\tau_l) \leq prio(\tau_h)\ \wedge\ dom(\tau_h) \not\leq dom(\tau_l) \quad (1)$$

for *Countermeasure I*, where T is the set of threads, $dom(\tau)$ is the clearance of the thread τ and $prio(\tau)$ is its priority. The predicate for *Countermeasure II* is:

$$p_{delay}(\tau_h, t) := \exists\, \tau_l \in T.\ prio(\tau_l) \leq prio(\tau_h)\ \wedge\ dom(\tau_l) \not\leq dom(\tau_h)\ \wedge\ max\_delay_l > 0 \quad (2)$$

For the purpose of information-flow secure scheduling, I propose to transform intransitive information-flow policies into transitive information-flow policies by restructuring servers at intransitive points and by introducing additional secrecy levels for the additional threads of these servers.

The use of static predicates allows off-line admission tests to consider the above two countermeasures. By characterizing the effect of *Countermeasure I* as a blocking term, a large class of existing admission tests can be reused to determine whether all real-time threads will meet their deadlines. This blocking term is the *prohibition time*:

$$bb_l^{pr} = \sum_{\tau_h \in T_{H+}} \left\lceil \frac{\Pi_l}{\Pi_h} \right\rceil .bb_h \qquad \textbf{where} \quad T_{H+} := \{\tau \in T | prio(\tau_l) \leq prio(\tau) \ \wedge \ p_{transitive}(\tau)\} \tag{3}$$

where Pi_h is the period and bb_h is the maximum blocking time of the higher prioritized thread τ_h.

The effect of *Countermeasure II* is negligible for the admission because the maximum time that a thread τ_l can executed non-preemptively (i.e., max_delay_l) is small compared to the budgets and blocking frequencies of higher prioritized threads.

From the prohibition time bb_l^{pr}, it is easy to see that τ_l can run whenever a higher prioritized thread τ_h blocks for which $p_{transitive}(\tau_h)$ does not hold. This ability to reap benefit of thread blocking times is the reason why the proposed fixed-priority scheduler can accept significantly more real-time threads than time-partitioning schedulers [ARI, Kop98], the state-of-the-art information-flow-secure real-time schedulers.

2.2 A Machine-Checked Non-Interference Proof

To prove the proposed budget-enforcing fixed-priority scheduler information-flow secure, I have constructed an abstract formal model of this scheduler and proved it to be non-interference secure.

Non-interference [GM82] is the prevailing formalization for the complete absence of security policy violating information flows. Given an arbitrary observer and two arbitrary initial states, which differ only in the states of threads that are higher classified than this observer, a scheduler is non-interference secure if the resulting schedules are indistinguishable for this observer.

The key insight, which has lead to the non-interference result for the proposed scheduler, is that the predicate $same\_high\_state$ over pairs of states is an invariant of the proposed scheduler. Informally, this predicate states the following two points:

1. The states of observable threads are identical in the two states of the pair; and
2. Threads, which are higher prioritized than an observable thread agree on the jobs they execute, on their remaining budgets, on the time that remains to their deadline, and on their thread state in the sense that they are either inactive in both states of the pair or active in both of these states.

The main result follows from observing that states for which $same\_high\_state$ holds are indistinguishable for the corresponding observer.

The formal model of the scheduler and the proof of non-interference have been formalized in the theorem prover PVS [ORS92]. The PVS sources and the machine-checked proof of the scheduler are publicly available [Völ10].

3 A Sound Security Type System for Low-Level Operating-System Code

The low-level operating-system code of microkernels and of the multilevel servers of open microkernel-based systems have a number of peculiarities, which make an information-flow analysis challenging:

- the peculiar ways in which multilevel servers interact with their clients and with the kernel;
- the side effects from interactions with the underlying hardware;

- the peculiar programming patterns, which combine C++, C and assembler in a way that is not always conform to the standard of these programming languages; and
- the lack of knowledge about the system's information-flow policy at the time when the kernel and the multilevel servers are analyzed.

To not risk overlooking the information flows these peculiarities involve, a sound information-flow analysis for low-level operating-system code must address all of these challenges.

In my thesis, I address the above challenges by first translating the to-be-checked operating-system code into an intermediate programming language — *Toy* — and by then checking the resulting *Toy* program together with interleaved executing side effects. These side effects are also *Toy* programs and which characterize the interaction with the underlying hardware and with separately checked components. A protection-parametric analysis with a universal lattice for shared-memory programs allows low-level operating-system code to be checked without precise knowledge about the usage scenario or its information-flow policy. In the following, I shall summarize this approach.

3.1 The Non-Deterministic Intermediate Programming Language *Toy*

Toy is a simple non-deterministic imperative programming language, which I have designed specifically for the purpose of analyzing low-level operating-system code and the side effects from the underlying hardware.

Toy inherits all interpreted data types and the semantics of most arithmetic operations from C++, although the formal semantics of *Toy* leaves most of the details of these types and operations abstract. In addition, *Toy* knows about bits, bytes, and words and about an address data type, which allows any bit in memory and in the processor registers to be addressed individually and in a unique fashion. Hence, *Toy* is based on a bit-granular memory model.

Because *Toy* facilitates a non-deterministic binary choice operator and parallel composition, the non-deterministic evaluation order of the value computations and side effects of C++ expressions can directly be expressed in *Toy*. The excessive use of non-allocated temporaries combined with non-deterministic choice allows for the consideration of compiler optimizations such as out-of-thin air values and stack- and register-allocation strategies.

The key property of *Toy*, which makes it suitable for an information-flow analysis of low-level operating-system code, is the clear separation of control-flow non-determinism and input non-determinism. In the formal PVS-based semantics of *Toy*, the latter is captured with the help of an almost arbitrary input oracle. There is only one constraint on any two input oracles used in the non-interference proof to produce the inputs for two runs of the program from observer indistinguishable initial states: the two oracles have to agree on the values they provide as updates for write-shared variables whose learned secret (see below) is dominated by the observer secrecy level.

3.2 Shared Memory, Locks, and Learned Secrets

Concurrently executing threads interact with the to-be-checked operating-system code through shared memory and other shared kernel or server object without necessarily being analyzed themselves. To not risk overlooking illegal information flows, we have to characterize the worst-case behavior of concurrently executing threads with regards to information leakage. In particular, we must verify that these threads cannot relay secrets through the to-be-checked operating-system code.

To verify the absence of leakage due to concurrently executing threads, I suggest in my thesis to keep track of the secrets these concurrently executing threads may learn from the to-be-checked program before and while this program executes. These secrets are the confidential data, which concurrently executing threads can access before the to-be-checked program starts executing, plus the secrets the to-be-checked program stores in externally visible regions of shared memory or in likewise accessible shared objects. While the to-be-checked program executes, concurrently executing threads may return an arbitrary encoding of the so far learned secrets with the intent to relay them with the help of the to-be-checked program.

To keep track of secrets concurrently executing threads learn from the to-be-checked program, I introduced a second set of dynamic secrecy levels — the *learned secrets* — and a corresponding typing rule for concurrently executing threads, which updates these learned secrets.

Notice, not all read-shared memory regions are visible at all points in time. Locks and certain precautions such as the disabling of interrupts in uniprocessor systems make these regions temporarily inaccessible, provided all concurrently executing threads adhere to the locking discipline of these regions. Hence, the proposed analysis can tolerate the temporary storage of confidential data in lock-protected shared-memory regions, provided that this imminent breach of confidentiality is repaired before the region becomes visible again. This tolerance was a crucial ability to prove correctness of Osvik's countermeasure against AES cache side-channel attacks.

3.3 A Sound Control-Flow-Sensitive Security Type System for *Toy*

The typing judgements of the security type system for *Toy* have the form

$$[l_{ip}, M_c] \vdash M^i,\ L^{i-1},\ i\ \{c\}\ M^{i+k},\ L^{i+k-1},\ i+k \tag{4}$$

where l_{ip} is the secrecy level of the context in which the *Toy* statement c is executed, M_c is the clearance of the physical addresses the to-be-checked program accesses during its execution and M^i, M^{i+k}, L^{i-1}, and L^{i+k-1} are the typing environments respectively the learned secrets before and after the evaluation of c. The typing rules for the expressions and statements of *Toy* are fairly standard with the exception of the following three points:

- after each atomic step of the to-be-checked program, the typing rules check whether those read-shared variables contain no secrets that are not protected by a lock. This is to detect leakages by the to-be-checked program in shared-memory variables that are later overwritten;
- after each atomic step, every typing rule invokes the update rules, which characterize the worst-case behavior of concurrently executing threads; and,
- every typing rule maintains and updates the secrets concurrently executing threads may learn.

A distinct feature of the proposed security type system is that it is only for the deterministic core of *Toy*. The special nature of the low-level operating-system code of microkernel-based systems is that the individual system calls typically terminate quickly. It is therefore feasible to check all possible ways in which the control-flow non-determinism in the resulting *Toy* program can be resolved, one at a time. Although the standard rules for non-deterministic choice [Sab01] would be sound, a separate analysis is much more precise.

The main soundness result follows from the fact that the typing rules check for dominated secrecy levels after each atomic step and that all statements preserve l-similarity over dynamic types. That is, given a typing environment M^i and two states s^i and t^i, then s^i and t^i are l-similar with respect to an l-classified observer if they differ only at higher than l classified addresses (i.e., $\forall a.M^i(a) \leq l \Rightarrow s^i(a) = t^i(a)$). The result that all statements are good in the sense that they preserve l-similarity over dynamic types, follows straight forwardly by structural induction.

3.4 From Type Checking to a Protection-Parametric Information-Flow Analysis

Although the above sketched security type system is sound, the results from applying it to the entire microkernel or to multi-level servers are not very interesting. In these results, the information flows of a multitude of operations and permission settings are blurred into one statement on the contained information flows. And, as yet, the results are for one specific information-flow policy.

To avoid the above complications, I propose in my thesis a universal lattice for shared-memory programs and a protection-parametric analysis of the individual operations of system calls or server invocations.

In my thesis, I extend Hunt and Sands' [HS06] idea to check programs with a universal lattice to shared memory programs. A secrecy level of the universal lattice is the set of all program-variable identifiers from which information may have flown into the such typed result. However, shared-memory variables can assume multiple secrecy levels over time. Hence, I had to extend the universal lattice by Hunt with version numbers for shared-memory variables. After checking the program with the universal lattice, the revealed information flows must be validated against the information-flow policy once this policy is known. For that, the variable identifiers are replaced by the secrecy levels of the information-flow policy and the least upper bound of all these levels is taken as the type of a result. The program is secure for the given policy provided the clearances of observers dominate all these secrecy levels.

To obtain results for the individual operations of a system call, I propose to fix certain crucial parameters as additional semantic information and to perform the respective analysis based on these parameters. Unknown objects, such as the invoked capability, are thereby replaced by placeholder objects. The such identified information flows constitute a leakage of confidential information if, in the concrete scenario, the leaked-over placeholder object instantiates to a shared kernel or server object and if the communication partners are both authorized to perform the respective operations.

4 Timing-Leak Transformations and Secure Resource Usage

Timing-leak transformations [Aga00] replace in a timing-insensitive information-flow-secure program operations, which exhibit a secrecy-dependent timing behavior, with semantically equivalent operations, which do not exhibit such a behavior. Although the thesis does not contribute in the area of timing-leak transformations. Such a transformation is required to eliminate the internal timing leaks that remain in successfully-checked multilevel components. Several approaches are mentioned including cross copying [Aga00], transactional branching [BRW06], and unification [KM07]. However, most promising for low-level operating-system code seems to be a transformation sketched by Warnier [BRW06], which relies on Engblom's [EES+03] worst-case execution-time (WCET) analysis: to defer externally-observable events to a safe upper bound of their worst case occurrence time.

In a sense, the proposed scheduler and the contention-leak avoidance of the below resource access protocol are instances of this transformation:

- the scheduler defers the scheduling of lower or equally prioritized threads to the worst-case point in time up to which a possibly leaking thread could influence these threads; and
- contention is made undetectable by requiring the resource acquiring thread to always provide the worst-case time it would need to obtain the requested resource.

The above two observations motivated the following investigation, which has lead to a secure real-time resource access protocol for uniprocessor systems called the *donation-ceiling protocol*. First, I summarize why one of the two forms of timeslice donation [SWH05] — downward donation — is secure when combined with the proposed budget-enforcing fixed-priority scheduler. Then, I recapitulate the donation-ceiling protocol, an alternative description of the basic priority-ceiling protocol [SRL90].

4.1 Timeslice Donation

A downward-donating thread forwards as part of its inter-process communication (IPC) both, its current priority and its time to the donatee. This donatee is either the receiver of the IPC call, which executes the donator's request or a request of another thread, or a thread to which this receiver is directly or indirectly donating.

To see why from an information-flow perspective it is safe to use downward donation in a system with the proposed budget-enforcing fixed-priority scheduler, we have to realize that synchronous reliable IPC enables bidirectional information flows between the communication partners anyway and that, unlike with upward donation, no other thread is affected by downward donation.

Downward donation reveals to the donatee the point in time when a threads starts a donating call and the amount of time it donates. Conversely, the donatee leaks to the donator how much donated time it consumes. Because downward donation forwards both the donator's time and priority, the donatee runs only on this donated time when the donator could consume this time. Therefore, if the donator must not leak information to lower or equally prioritized threads, *Countermeasure I* of the proposed scheduler prevents their execution until the budget of the donator is consumed. Whether this budget is consumed by the donator, by a donatee or by a budget-consumer thread is thereby irrelevant from the perspective of those lower or equally prioritized threads that must not receive information from the donator.

4.2 Donation-Ceiling Protocol

The donation-ceiling protocol mimics the basic priority-ceiling protocol [SRL90] by accumulating the resource acquiring threads at so called *ceiling threads*. The protocol uses such a ceiling thread for each distinct priority-ceiling level of the resources of the system.

In order to acquire a resource, a thread must request this resource with a downward donating call from the ceiling thread with the lowest associated ceiling priority that is still higher than its own priority. This ceiling thread in turn handles the request itself if the resource has a priority ceiling that is equal to or lower than the priority-ceiling level for which it was created. Otherwise it forwards the request to the ceiling thread that is responsible for the next higher priority-ceiling level. Thereby, downward donation ensures that the resource holder always runs at the time and priority of the highest prioritized thread that requests a resource with the same priority ceiling.

The equivalence of the donation-ceiling protocol and of the basic priority-ceiling protocol follows from a comparison of the rules of the respective protocols.

Because downward donation does not affect unrelated thread when used in combination with the proposed budget-enforcing fixed-priority scheduler, no information can be leaked to threads that do not acquire resources. Information leakage due to resource contention is avoided by deferring any execution, which follows a resource access, to the point in time that resembles the worst-case resource access time. For the basic priority-ceiling protocol and hence also for the donation-ceiling protocol this worst-case resource access time is twice the worst-case time that a thread holds the resource [SRL90].

5 Case Studies and Osvik's Countermeasure Against AES Cache Side-Channel Attacks

Although the development of an efficient type-checking tool has been out of the scope of my thesis, I have exemplified the applicability of the proposed analysis in three case studies: a page-table walk, L4-IPC and a presumably secure buffer-cache server. In addition, I have taken advantage of the description of hardware side effects as interleaved-executing *Toy* subprograms to prove Osvik's countermeasure against AES cache side-channel attacks [OST05] correct. That is, Osvik's countermeasure protects the key, the plaintext and intermediate encryption results against leakage over the processor caches. To my best knowledge, this is the first security-type-system-based proof of such a countermeasure.

5.1 Page-Table Walk

The first case study exemplifies the information-flow analysis of low-level operating-system code with hardware side effects. Virtual-memory accesses involve such a side effects if the virtual-to-physical address translation is not cached in the translation lookaside buffers of the CPU. It traverses the page tables, performs various access-right checks, and updates the accessed and dirty bits in the used page-table entries.

Given an implementation of this hardware side effect as a *Toy* subprogram, the security type system for *Toy* can check the virtual memory access for security policy violating information flows by checking both the *Toy* program, which results from translating the respective C++ memory access, and the interleaved executing hardware side effect.

The analysis of a size-aligned virtual-memory read correctly revealed that information about the context in which this access is executed is leaked to the accessed bits of the used page-table entries and that the result of the address translation reveals information about the code-segment privilege level, about the permission bits and about page- and page-table pointers in the used page-table entries.

5.2 L4-IPC

The second case study exemplifies the protection-parametric analysis of a system call of one of the L4-family microkernels: Nova's IPC send operation [Ste09].

The analysis correctly revealed the covert storage channels of this operation. To avoid blurring the result with other system calls and other operations, the parameters were chosen to select an authorized data-word-only IPC send operation with send timeout zero. The analysis required 6 placeholder objects: a TCB and UTCB for the sender and for the receiver, an IPC gate, which refers to the receiver, and a capability, which refers to the IPC gate.

The application of the universal lattice for shared-memory programs allows the results of this analysis to be reused for arbitrary settings. In such a setting, the variable identifiers for the input parameters of the system call, which are leaked to the message registers and output parameters of the receiver, are replaced by the secrecy levels of the information they hold to see whether the information flows violate the system's security policy.

5.3 Buffer-Cache Server

The third case study combines all results of this thesis in an analysis of a presumably secure buffer-cache server. A buffer cache stores recently accessed file blocks in per-client memory pools while facilitating a safe sharing of buffers between differently classified clients.

The invocation of the multi-threaded buffer cache server, the invocation of the underlying secure file system, and the response of this server make use of the protection-parametric analysis of system calls of the microkernel respectively of the file-system functionality. The accesses to the data structures for maintaining cached file blocks, which are typically highly optimized and therefore difficult to free from covert channels, are synchronized with the help of secure resource access protocols, which in turn depend on the proposed scheduler to avoid external timing channels. And finally, an application of the universal lattice for shared-memory programs allows the results of the analysis of the buffer-cache server to be reused in various scenarios, including in analyses of clients of this server.

Although the analysis correctly identified the buffer-cache server as timing-insensitive non-interference secure, a flaw in the L4 capability revocation mechanism prevents a safe sharing of buffers between differently-classified clients. In the current interface of L4, the amount of capabilities, which have to be traversed when a capability is revoked, cannot be bounded from above. Therefore, the timing leaks, which origin from varying this amount, cannot be transformed out. The thesis gives directions for avoiding these leaks, however, a thorough discussion of information-flow secure capability revocation mechanisms is left for future work.

5.4 Osvik's AES Countermeasure

To speed up the encryption, many performance-oriented implementations compute the arithmetic operations of AES with the help of in-memory lookup tables. By measuring the memory access times of previously loaded preparation data, adversaries can deduce the encryption key from the cache conflict misses the key-dependent table lookups cause on the preparation data. Osvik, Shamir, and Tromer [OST05] propose several countermeasures against these cache side-channel attacks. One accesses the lookup tables with cacheline stride after each encryption round. This way, an adversary will find the entire table accessed.

To prove this countermeasure correct with the help of the security type system for *Toy*, the hardware side effect of the cache has to be implemented as an interleaved executing *Toy* subprogram and checked together with the implementation of AES. To do so, I introduce an artificial hardware register, which contains one bit per cacheline. A set bit indicates that the caching of the preparation data is unaltered. A cleared bit indicates that a memory access of the checked AES implementation has possibly replaced preparation data. To characterize the cache replacement strategy, every memory access of the checked AES code is therefore complemented with a hardware side effect, which sets the corresponding bit in the artificial cache register. An analysis of this complemented code correctly reveals a possible leakage of the $high$-classified key by raising the secrecy level of the cache bits to $high$ after each encryption round. Would an adversary be able to read the cache bits in between the encryption round and the countermeasure, it could reveal the key bits. However, because I assume the round and the countermeasure to execute non-preemptively, the shared cache bits are protected by a suitable lock. The countermeasure accesses the lookup table with secrecy-independent indices. As a consequence, the secrecy levels of the cache bits drop to low. The imminent breach of confidentiality is prevented, which proves Osvik's countermeasure correct.

6 Conclusions

In this document, I have summarized the challenges and contributions of my doctoral thesis: "Provable Protection of Confidential Data in Microkernel-Based Systems". The major contributions of this thesis are:

- an analysis of scheduling-related timing channels in fixed-priority schedulers;
- a provably non-interference-secure budget-enforcing fixed-priority scheduler;
- a non-interference-secure real-time resource-access protocol;
- a protection-parametric analysis method of shared memory programs;
- a sound security type system for low-level operating-system code of microkernel-based systems; and
- the first security-type-system-based proof of a countermeasure against cache side-channel attacks.

Future work is left in many areas, in particular, in efficient type-checking tools for low-level operating-system code, language extensions for protection parametric analyses, construction guidelines for non-interference-secure multilevel servers, and the elimination of the identified leaks in L4's capability revocation mechanism and in the current implementation of downward donation.

References

[Aga00] J. Agat. Transforming out Timing Leaks. In *ACM Principles of Programming Languages*, Boston, Massachusetts, Jan 2000.

[ARI] ARINC. *ARINC 653-1 Standard*.

[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).

[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.

[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.

[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.

[Här02] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[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.

[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.

[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.

[Hu92] W. Hu. Lattice Scheduling and Covert Channels. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1992.

[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.

[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.

[Sab01] 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.

[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.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. *IEEE Transaction on Computers*, 39, 1990.

[Ste09] Udo Steinberg. *NOVA Microhypervisor Interface Specification*. Technische Universität Dresden, Dresden, Germany, December 2009. available at http://hypervisor.org.

[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.

[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.

[Völ10a] Marcus Völp. PhD thesis - PVS sources. available at http://os.inf.tu-dresden.de/˜voelp/sources/thesis/index.html, 2010.

[Völ10b] Marcus Völp. *Provable Protection of Confidential Data in Microkernel-Based Systems*. PhD thesis, Technische Universität Dresden, Dresden, Germany, August 2010.