TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports

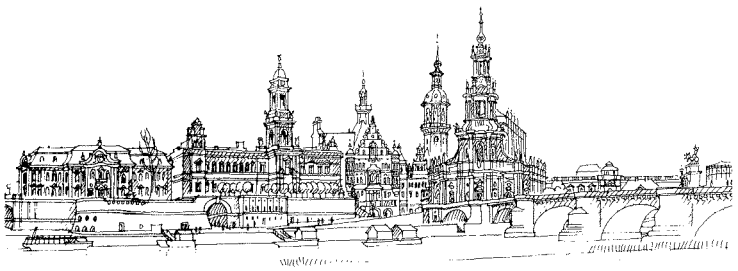
ISSN 1430-211X

TUD–FI02–03–März 2002

Michael Hohmuth, Hendrik Tews, and Shane G. Stephens

VFiasco Project

Applying source-code verification to a microkernel — The VFiasco project



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

Applying source-code verification to a microkernel — The VFiasco project

Michael Hohmuth
Hendrik Tews

Dresden University of Technology
Department of Computer Science

Shane G. Stephens

University of New South Wales
School of Computer Science and Engineering

vfiasco@os.inf.tu-dresden.de

Abstract

Source-code verification works by reasoning about the semantics of the full source code of a program. Traditionally it is limited to small programs written in an academic programming language. In this paper we present the VFiasco (Verified Fiasco) project, in which we apply source-code verification to a complete operating-system kernel written in C++. The aim of the VFiasco project is to establish security relevant properties of the Fiasco microkernel using source code verification. The project's main challenges are to develop a clean semantics for the subset of C++ used by the kernel and to enable high-level reasoning about typed data starting from only low-level knowledge about the hardware. In this paper we present our ideas for tackling these challenges. We sketch a semantics of C++ and develop a type-safe object store for reasoning about C++ programs. This object store is based on a hardware model that closely resembles the IA32 virtual-memory architecture, and on guarantees provided by the kernel itself.

Contents

1 Introduction **1**

2 Related work **2**

3 A semantics of C++ **3**
3.1 State transformers 3
3.2 Typed data 5

4 A type-safe object store **5**
4.1 Programmer's expectations and system guarantees 6
4.2 Hardware model 6
4.3 Verification environment 7
4.3.1 Encapsulating system guarantees . . 7
4.3.2 The object-store layer 8
4.4 Base Assumptions 9

5 Conclusion **9**

1 Introduction

The VFiasco project aims at the mechanical verification of security-relevant properties of the L4-compatible Fiasco microkernel [12].

The goal of the project is an operating-system kernel that provides *verified* security guarantees. Such a kernel could be used as a basis for building applications with high-level security requirements. Verification is very expensive (both in man power and time); for success it is crucial to minimize the size of the system. Huge bug-afflicted monolithic kernels are outside the scope of current verification technologies. On the other hand, microkernels are the smallest kernels that provide an anchor for building secure systems: separate protected address spaces. Therefore, they are the best choice for constructing a verified secure system.

VFiasco is a work-in-progress. In this paper we report in detail on one aspect of the project: the modeling of a type-safe object store on top of a model of virtual-memory hardware. Further, we outline the semantics of C++ that we will use in the verification.

To our knowledge, the VFiasco project is unique in scope and intended thoroughness. We aim at modeling all of the kernel's source code in very fine grain, and we intend to "run" this software model on a hardware model that closely resembles real hardware. These qualities are meant to establish an as-yet unseen level of confidence in our software. Our formal-verification approach exceeds even what is necessary to fulfill the development requirements of the Common Criteria's[1] highest assurance level, EAL7.

Fiasco has been implemented in C++. For the verification we develop a dialect of C++ with a precise semantics, which we call "Safe C++." The verification will be carried out in the interactive theorem prover Isabelle/HOL [21]. This theorem prover uses higher-order logic (HOL) as its input language. Therefore, we translate the kernel's source code from Safe C++ into its semantics expressed in HOL. In our approach, a *logic compiler* performs this translation

This research was supported by the Deutsche Forschungsgemeinschaft (DFG) through DFG grant Re 874/2-1. Shane G. Stephens was supported by a Study Grant from Deutscher Akademischer Austauschdienst (DAAD) through the International Quality Network (IQN) program "Rational Mobile Agents and Systems of Agents."

[1] The "Common Criteria for information Technology Security Evaluation" (CC; ISO 15408) replaces the Trusted Computer System Evaluation Criteria (TCSEC; better known as the "Orange Book") in the U.S.A. and Information Technology Security Evaluation Criteria (ITSEC) in the E.U.

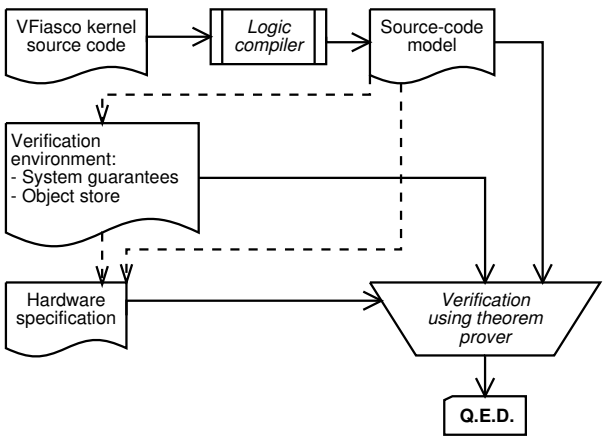


Figure 1: The verification process. Legend: Solid arrows show the flow of data. Dashed arrows indicate a ≪uses≫ relationship.

automatically. This technique is in stark contrast to approaches in which parts of the source code are translated manually to a more or less abstract model. Figure 1 illustrates our verification methodology.

The basis of the semantics of Safe C++ is a model of the computer system, which we must provide in the theorem prover. An important problem in the project is to find the right abstraction level for this model. To facilitate the verification, we would like to have the abstraction level of a virtual machine that provides a *type-safe object store*. Under a type-safe object store we understand an abstract model of memory that supports reading and writing of typed values and that guarantees safe accessibility of these values. A type-safe object store would allow us to reason on a comfortably high level, ignoring the complexity of contemporary virtual-memory systems and memory allocation.

However, we cannot simply assume such an object store before verifying the Fiasco microkernel. Fiasco executes in a much more hostile environment—on virtual-memory hardware. In fact, one of the kernel's tasks is the provision of guarantees that allow the construction of such an object store in the first place. Therefore, the existence of an object-store layer with strong properties should be a proof goal, not a base assumption.

In this paper, we fill the gap between high-level programming languages (in our case Safe C++), which provide safety by means of protecting typed memory objects from arbitrary accesses, and contemporary hardware with virtual memory. We develop a type-safe object store based on a set of memory models that mimic the way a high-level–language programmer thinks of memory, but still can be implemented using a concrete CPU model. Using these memory models, it is possible to reason about the Fiasco kernel, ignoring the current virtual-memory setup and the effects of page faults on the program state.

This paper is organized as follows. In Section 2, we discuss previous work in the field of operating-system verification, and we explain why we use theorem provers and not model checking. Section 3 outlines the semantics of C++ that we use in the verification. Section 4 contains the main part of this paper. In this section, we explain what can be expected from an object store, and we derive design goals for our model. We briefly discuss our hardware model and then develop a verification environment for system-level software, including a type-safe object store. We conclude the paper with a summary in Section 5.

2 Related work

Our research is related to the following work areas: theoretical work on program verification, model checking, proof-carrying code, static source-code checking, object-code verification, and theorem proving. In this section, we discuss these categories in turn.

Theoretical work. Theoretical accounts sometimes use examples from operating systems. For instance an algorithm for mutual exclusion is verified in all of [5, 1, 18, 22, 15]. The theory described in these papers is often highly relevant for our work. However, the methodology used there for small (and clean) examples cannot be applied to the Fiasco microkernel.

Model checking. Model checking has been successfully applied to several systems [4, 2, 23]. When using model checking one first has to reimplement some parts of the system within the model checker. During this translation one has to abstract from details that do not affect the result of the verification. This is necessary because model checking has inherent restrictions, especially with respect to the state-explosion problem. These two necessary steps—reimplementation and abstraction—limit the conclusions that can be drawn from model checking: It is impossible to verify that the model faithfully represents the source code.

For instance, in [23] Tullmann and colleagues verify liveness properties of the Fluke kernel's [9] IPC subsystem. Thereby they abstract away from the actual data that is transmitted. While they actually proved the absence of deadlocks, it is theoretically possible that the IPC subsystem deadlocks because it dereferences a malicious user pointer (which has been abstracted away in the model checker).

In our project we will model the complete source code of the Fiasco kernel, including the page fault handler. If we manage to prove that the IPC system calls do always terminate, then this proof necessarily includes a (sub-)proof about the correct handling of user pointers. While model checking can be applied with almost no user interaction, our approach requires highly skilled staff operating the theorem prover.

Proof-carrying code. Proof-carrying code [20, 19] solves the problem of executing untrusted (user-supplied) code in kernel mode. In this approach the kernel (developer) publishes a security policy, and every extension to be downloaded must be accompanied with a formal proof showing that the extension conforms to the security policy.

Checking the validity of the proof can be done efficiently in the kernel because proof checking is closely related to type checking [10]. The author of the kernel extension is responsible for the difficult part—constructing the proof and thereby verifying the extension. However, for a typical application of proof-carrying code (for instance a network filter) the involved verification is trivial and can often be done automatically.

In the VFiasco project, we tackle a rather different problem: proving the kernel *itself* correct. The problem of safely extending this kernel is orthogonal to our work.

Microkernels such as Fiasco are extended using user-level servers that run in their own address spaces. Some of these servers are an integral part of the system and must be trusted by all applications. Trusted servers need to be verified separately, independent of the underlying microkernel. Verification of user-level components and of such trusted servers is outside the scope of this work. We are undertaking the first and most basic step of system verification—proving the kernel correct.

In general, applying verification to a microkernel based system is much easier than applying it to a traditional monolithic system. The reason is that the enforced address-space separation of system components in a microkernel based system makes it possible to verify the components independently from each other.

Static source-code checking. There are many tools in the spirit of `lint` that statically analyze source code. Some of these provide customizable rule sets and have successfully been used to find bugs in operating systems [8, 6, 7].

Static source-code checking is different from testing in that it analyzes the source code instead of running it. With testing it has in common that it assists in finding programming errors. In the VFiasco project our concern is not so much to find errors, but to *give guarantees* about their absence.

Object-code verification. One problem of source code verification is that, for a compiled language, its results do not apply to the running system: The compiler might have changed the program such that the running system violates properties that have been proven before. One way out of this dilemma is to verify the object code itself as proposed in [25]. The main problem with object-code verification is that it is much more difficult. The examples handled in [25] have only a fraction of the size of successful source-code verification projects like for instance the Huisman's and colleagues' Java-Vector verification [14]. Further, also with object-code verification absolute security remains an illusion: The theorem prover used in the verification might be unsound, thus allowing one to prove arbitrary results.

We believe that source-code verification in combination with a well-tested compiler provides enough security for all practical purposes.

Theorem proving. Related work that comes closest to our project is those that applies theorem proving to source code and more especially to operating-systems source code.

In [17] Liu and colleagues use the theorem prover Nuprl [3] in the Ensemble project to verify the correctness of network-protocol stacks and to optimize such stacks. They use several tools that translate their specifications and the Ensemble source code into Nuprl and back. In the VFiasco project we also plan automated translation of source code into the theorem prover Isabelle. However, there are two important differences. First, to enable the verification in the Ensemble project the original C source code was rewritten in a carefully chosen subset of the functional language Ocaml [16]. Features of Ocaml that are more difficult to handle, like objects or exceptions, are not used in Ensemble.

In contrast, we plan to develop a semantics of a subset of C++ that essentially contains everything needed for kernel programming, including abrupt termination[2], `longjmp`'s, and pointer arithmetic. Second, Liu and colleagues did *not* verify the source code. Instead, they verified program transformations.

Our approach to a semantics of C++ is very similar to the one used in the LOOP project for Java [13]. We also use coalgebras to represent statements and expressions. In the LOOP project Jacobs and colleagues focus on the verification of Java applications. Consequently they use an object memory that directly represents Java objects [24]. A central aim of the VFiasco project is to incorporate system internals like page fault handling and protection levels into the verification. Therefore we need a more low-level view on the object memory.

3 A semantics of C++

This section outlines the semantics of C++ that we need for the verification of the Fiasco source code. The approach we take here is very pragmatic: For every feature of C++ we assess both the difficulty of defining a semantics for that feature and the difficulty of writing the Fiasco kernel without that feature. The result of that assessment defines the language Safe C++, which is used to implement Fiasco. Although C++ has a reputation of being an unsafe and dirty language, it is surprising to see how much of C++ can be modeled without difficulty: Safe-C++ will allow pointer arithmetic and all of C++'s flow control structures including `break`, `continue`, `setjmp`/`longjmp`, and even `goto`.

An important goal in the design of Safe C++ is that it shall be compatible with C++ modulo some preprocessor directives and a small library (encapsulating direct hardware access). If this goal is met we can use a standard C++ compiler for Safe C++ programs.

3.1 State transformers

In the semantics of C++ we do not distinguish between statements and expressions as the C++ grammar does:

[2] An expression or statement terminates *abruptly* if the control flow does not reach the end of the statement or expression because, for instance, a `break` or `return` was executed.

statements are considered as expressions of type void, where void is a type that contains precisely one (uninteresting) element $*$. The evaluation of such generalized expressions depends on the current state of the whole system. Therefore the semantics $[\![e]\!]$ of an expression e is a function that takes the current state of the system as input. Let St be the set of all possible system states. We will elaborate on this set in Section 4.2, for the moment we can ignore the details.

When evaluation an expression e in a state s there are three fundamentally different possibilities:

- The expression returns *normally* and delivers a result (of $*$ with type void in case e corresponds to a C++ statement). In addition the evaluation might have changed the current state. Therefore the semantics of e also returns a successor state $s' \in \mathsf{St}$. The states s and s' differ, if e causes side effects, for instance by executing assignments, but also if e causes a page fault that is successfully handled by the current page-fault handler.
- The expression terminates *abruptly* with an *abnormal* result because it does a longjump or (for statements) executes a `continue` or a `break`. An abnormal result consists (roughly) of a tag (for distinguishing breaks from continues), a state, and possibly some additional information. Special statements (like while loops) can catch abnormalities, extract the state, and continue execution normally.
- The evaluation does not terminate or some catastrophic event, like asserting false, happens.

We capture this possible behavior with the following (pseudo) Isabelle type definition:

```
datatype α ExprResult =
    Normal   "α" "St"
  | Abnormal "St AbnormalResult"
  | Bug
```

Here α is a type variable that gets instantiated by the concrete type of the expression in question. The Isabelle `datatype` is similar to the variant records of the programming language Modula-2: An element of α `ExprResult` is tagged with either `Normal`, `Abnormal`, or `Bug`. If it is tagged with `Normal` it carries an element of α and one of St, if it is tagged with `Abnormal` it contains an element of St `AbnormalResult`, a type which we discuss below. The first proof obligation in the VFiasco project will be that the result `Bug` does never occur.

With the preceding type definition the semantics of an expression e of type α is a function (in the mathematical sense)

$$[\![e]\!] : \mathsf{St} \longrightarrow \alpha\ \texttt{ExprResult}$$

Such a function with a structured codomain is usually called a coalgebra. Here we use the term *state transformer* to denote such a function.

Sequential composition of statements (in the source code) is mapped to composition of state transformers (in the theorem prover). The composition of two state transformers is a higher-order function that combines two state transformers and returns a new state transformer. It is defined as follows:

$$\begin{array}{rcl}
\big([\![e_1]\!] \circ [\![e_2]\!]\big)(s) & = & \texttt{cases}\ [\![e_1]\!](s)\ \texttt{of} \\
\quad \texttt{Normal}(r, s') & \Rightarrow & [\![e_2]\!](s') \\
\mid\ \texttt{Abnormal}(a) & \Rightarrow & \texttt{Abnormal}(a) \\
\mid\ \texttt{Bug} & \Rightarrow & \texttt{Bug}
\end{array}$$

This definition uses pattern matching on the result of $[\![e_1]\!](s)$: In case the first expression returns normally the result r is discarded and the expression e_2 is evaluated on the intermediate state s'. If e_1 does not return normally then e_2 is not evaluated at all, as expected. However, a surrounding block can catch the abnormality.

There is no space here to discuss all possible reasons for abrupt termination. Let us do just one example: The `continue` statement. The type `AbnormalResult` is a data type similar to `ExprResult` that lists all possible abnormalities:

```
datatype γ AbnormalResult =
    Continue  "γ"
  |   ...
```

The semantics of the `continue` statement is now rather simple:

$$[\![\texttt{continue}]\!](s) \quad = \quad \texttt{Continue}(s)$$

Note that the type variable γ gets instantiated with St here. Continue abnormalities are caught at the end of the body of `for` and `while` loops with the following function `CatchCont`:

$$\begin{array}{l}
\texttt{CatchCont}([\![e]\!])(s)\ =\ \texttt{cases}\ [\![e]\!](s)\ \texttt{of} \\
\quad \texttt{Abnormal}(a)\ \Rightarrow \\
\qquad (\ \texttt{cases}\ a\ \texttt{of} \\
\qquad\quad \texttt{Continue}(s')\ \Rightarrow\ \texttt{Normal}(*, s') \\
\qquad\quad \mid\ x\ \Rightarrow\ \texttt{Abnormal}(a)\ ) \\
\quad \mid\ x\ \Rightarrow x
\end{array}$$

For the semantics of a loop we need the iteration of a state transformer:

$$\begin{array}{rcl}
\texttt{iterate}(e, 0)(s) & = & \texttt{Normal}(*, s) \\
\texttt{iterate}(e, n+1)(s) & = & \big(\texttt{iterate}(e, n) \circ [\![e]\!]\big)(s)
\end{array}$$

To get the semantics of a while statement, one considers the following composite:

$$\texttt{iterate}\big([\![\mathit{cond}]\!] \circ \texttt{CatchCont}[\![\mathit{body}]\!],\ k\big)\ \circ\ [\![\mathit{cond}]\!] \qquad (\dagger)$$

If there exists a natural number k such that (†) returns either abnormally (because of a `break` or a `goto`) or normally with result false then the semantics of the while statement is precisely (†) with the least such k substituted. If there is no such k then the while loop does not terminate and its semantics is `Bug` $\in$ St `ExprResult`.

3.2 Typed data

In this subsection we discuss how to model typed variables and pointer arithmetic. For that discussion it is necessary to know a bit more about how we model the state of the system: The main ingredient of a state is the main memory, which supports as basic operations reading and writing of sequences of bytes.

However, for the semantics of C++ variables we need operations that read and write typed values. These operations should have the following property: Let v be a variable of some type t. In the main memory there will be some bytes that represent the value of v. If this memory area is modified (as a result of a bug) by writing a value of a type different from t then trying to read the value of v should result in an undetermined value. We solve this problem by using *underspecified functions*. A function is underspecified if the result for some arguments is not completely determined.

For example, to operate with C++ variables of type boolean we declare two functions in Isabelle:

```
byte_to_bool : byte list ⟶ bool
bool_to_byte : bool ⟶ byte list
```

We make two additional assumptions: First, we assume that the length of the list `bool_to_byte`(b) is fixed for all booleans b. Second, the following equation must hold for all possible b:[3]

$$\texttt{byte_to_bool}(\texttt{bool_to_byte}\ b) \quad = \quad b \qquad (\ddagger)$$

Note that we assume neither that a boolean is encoded in one byte (as the C++ standard does *not* prescribe `sizeof`$(\texttt{bool}) = 1$) nor that `false` is represented as 0.

Logically the use of an underspecified function amounts to universal quantification: The inferred results apply to all pairs of functions `byte_to_bool` and `bool_to_byte` that fulfill the assumptions. Or, to put it differently, our verification results apply to all C++ compilers, regardless of how booleans are represented in memory.

Writing a value to the boolean variable v is now a two stage process: First the function `bool_to_byte` transforms the value into its byte representation; then these bytes are written at the location of v. The assumption ($\ddagger$) guarantees that reading a boolean value at the location of v gives the same value back—as expected. However, if an integer is written at the location of v then it is impossible to infer something about the value obtained when reading a boolean from the location of v. Such a bug will typically produce an unsolvable proof obligation (unless the variable v is never used again), which means that the verification cannot be completed until the bug is fixed.

Note that with this approach of using underspecified functions we can also model C++'s pointer arithmetic: In the theorem prover the pointer arithmetic is performed according to the C++ standard yielding as result some location l in the memory. If the type of the pointer does not match the type of the data at l then some arbitrary value is produced, as discussed before.

In our semantics of C++ the locations of variables will also be underspecified. This means that the semantics does only allow to infer that the location of an automatic variable is above the stack pointer and that different automatic variables do not overlap—but not more. As a result, an "of by one" error in a stack-allocated array invalidates all other automatic variables (because it is impossible to prove that the locations of these other variables are different from the wrong pointer).

We have not yet decided about how to treat C++'s arithmetic on integer types. For instance in C++ we have $0 - 1 > 0$ for unsigned types while in the theorem prover Isabelle $0-1 = 0$. A correct semantics of C++ would have to define arithmetic modulo 2^w for unsigned types of width w. However, the special effects of arithmetic modulo 2^w are never used in the Fiasco kernel. So for the verification it might be more economical to treat arithmetic for integer types as partial functions, which are only well defined if the result is within the bounds. This latter approach would require to prove that all arithmetic in the kernel obeys the size limitations.

Safe-C++ will not contain floating point arithmetic. The Fiasco kernel does not use floating point arithmetic; besides, an exact formalization of the IEEE floating point standard in a theorem prover is a major project in its own [11].

Safe-C++ will probably not allow exceptions. Modeling exceptions with the state-transformer approach is no problem at all. However, exceptions are not used in Fiasco (mainly because they require a heavy library).

4 A type-safe object store residing in virtual memory

In this section, we discuss what a Safe-C++ program's state St contains and which operations it supports. This interface comprises the "architecture" for which our logic compiler produces "code."

It is possible to apply the state-transformer approach from Section 3 to environments with widely differing abstraction levels. In the VFiasco project our goal is to keep a high-level–language programmer's view during verification while still enabling reasoning about low-level hardware manipulation.

As an example, consider a safely-typed[4] object-oriented language such as Java. In such a language, a program's state consists of a global object store in which each typed object is referenced using a global index. This model has been successfully used in the LOOP project for modeling a Java object store [24].

Unfortunately, a storage model that is *a priori* type safe is not adequate for modeling a kernel environment for two reasons. First, such an assumption might be wrong—

[3] To ensure consistency we prove that there exist two functions `byte_to_bool` and `bool_to_byte` with these properties. For the type of booleans this is obvious, but how about the type defined by `typedef int huge[1000][1000][1000]`?

[4] By *safely-typed programming language* we mean a language in which all type errors can be detected at either compile time or run time. According to this definition, Java is safely typed, but C++ is not.

invalidating all verification results—because there is no system component that provides type safety. In the real world, the kernel runs on top of an untyped virtual memory and must ensure its own type safety. Second, kernel programmers sometimes need to circumvent the compiler's type safety for low-level systems programming, for example for manipulating CPU data structures. The power to do so is missing from safely-typed languages; this is why kernel programmers often choose C or C++ instead of Java or Modula-3.

The remainder of this section is organized as follows. In Section 4.1, we elaborate on the missing link between low-level virtual-memory hardware and the C++ object model, and we derive design goals for our storage model. Section 4.2 introduces the hardware model we use for verifying the Fiasco kernel. Section 4.3 explains our verification environment and Section 4.4 lists our base assumptions.

4.1 Programmer's expectations and system guarantees

Programmers of high-level languages such as Safe C++, including kernel programmers, make many assumptions about the environment in which their program eventually runs. For example, programmers assume that a program can successfully access objects that have been properly allocated (statically, on the heap, or on the stack). Table 1 lists a number of such assumptions.

During verification, it is advantageous to have access to a high-level type-safe object store and to have Table 1's assumptions available as known properties of that object store (henceforth called *object-store properties*). Recall that we use an interactive theorem prover to reason about programs; in other words, a human user operates the theorem prover. This user would like to reason on the level of the Safe-C++ programmer, and therefore needs to use facts the programmer originally assumed.

As stated before, the *a priori* assumption of these object-store properties would make our verification project meaningless. The point of the VFiasco project is to show that the Fiasco kernel works according to its specification based on much more low-level knowledge. We want to assume only very basic facts about the hardware and the Safe-C++ compiler. Additionally, we need to be able to circumvent the object store and access the hardware layer directly.

Therefore, instead of assuming object-store properties from the start, our approach is to prove them starting from low-level knowledge.

In summary, we aim for the following design goals in modeling our object store:

Credibility. We want to start only from very basic low-level assumptions. Therefore, the storage model should be based on a memory model that closely resembles the virtual-memory hardware on which the kernel executes. Further, we must document all base assumptions that we make about the hardware and the Safe-C++ compiler. The hardware model and the base assumptions are discussed in Section 4.2 and Section 4.4, respectively.

Type-safe object store. Efficient interactive reasoning about a program requires high-level knowledge of the program's state. Therefore, we need to create a verification environment that provides a type-safe object store with proven object-store properties. This environment consists of a mapping of an object-store interface to a virtual-memory interface. Section 4.3 describes our verification environment.

Direct hardware access. It must be possible to circumvent the object store and access virtual memory directly. We address this requirement in Section 4.3.2.

There are also a number of second-level design goals:

Reusability. The object-store specification needs to be generic enough to serve as the general target language of the logic compiler. Fiasco's high-level *and* low-level kernel code as well as boot code should be expressible. In the future, we also would like to use it as a target for user-program code. Section 4.3.1 explains how we achieve this goal.

Automation. Based on the object-store properties, we provide powerful theorem-rewriting rules that automatically simplify logic-compiled source code without operator intervention as far as possible. We discuss our rewriting rules in Section 4.3.2.

4.2 Hardware model

The hardware model provides the basis for the semantics of Safe C++. It defines the set of system states `St` and primitive operations, like reading in memory and inserting page mappings. A complete model of the Intel IA32 architecture is far beyond our project. Rather, we use an abstraction of the hardware that contains just those primitive operations that are necessary to run the Fiasco microkernel. In particular, hardware features such as real address mode, V86 mode, the floating-point coprocessor, segments, and so on are only modeled in a rudimentary way. For instance, the semantics of resetting the PE flag (which switches the processor into real address mode) is simply the special value `Bug`. This way it becomes a proof obligation that the PE flag is never reset.

During verification, translated microkernel code actually "runs" on the model. This ensures that several important proof obligations (e. g., the microkernel does not cause recursive page faults) are generated automatically by Isabelle.

The model currently consists of four main components:

- The physical memory is modeled by a specification that encapsulates reading and writing to memory.
- The TLB specification encompasses three TLB-related operations: insert, retrieve and flush.
- Page-fault handling is modeled by a fully-specified page-table–lookup function. In systems with software-loaded TLBs, this function is part of the operating system and must be verified by the techniques described in this paper.

Assumption (object-store properties)	**Reality** (low-level knowledge)	**Implied system guarantee**
All program code and properly allocated data are accessible	Any memory access can fault during a TLB or page-table access	Pinned memory, or kernel faults in "correct" memory; kernel is mapped into all address spaces
Reading after writing returns the value previously written; objects do not change value unless updated explicitly	different objects might overlap; the same object might be mapped twice	All objects are allocated such that no two object's virtual-address regions overlap
Program reads and writes typed objects	Objects are stored in byte sequences; the byte representation of most data types is unknown to the programmer	There exist two inverse functions that convert between typed values and byte sequences
Program operates in flat virtual address space	Program code and data are split into pages, some of which are stored noncontiguously in physical memory, and some of which are not memory-resident	Page-fault code and virtual address space maintain "illusion" of flat address space
A program's code is immutable	Kernel can change all programs' code, including its own	Kernel does not modify program code
Hardware interrupts do not change the program's state arbitrarily	CPU switches to different context and executes interrupt handler in kernel mode	Interrupt handlers do not modify memory except for a small set of explicitly declared "volatile" objects

Table 1: Examples of high-level–language programmer's assumptions and guarantees needed from the memory subsystem. Usually, programmers assume object-store properties like those in the left column. However, these properties are not true in general. In reality, facts like those in the middle column can falsify the assumptions. The right column shows properties that, when maintained by the runtime system, imply the object-store properties.

- Functions for reading and writing to virtual memory complete the specification. These functions capture the behavior of the MMU, and make use of the physical-memory and TLB specifications, as well as the page-table lookup function.

In the future this model will be extended with specifications of other CPU features such as privilege levels, input–output, and interrupts as required.

4.3 Verification environment

In this section, we construct a type-safe object store, assuming only a model of virtual-memory hardware.

4.3.1 Encapsulating system guarantees

System specifications. We have been able to prove the object-store properties by assuming the properties of Table 1's "implied system guarantee" column. As a means for structuring the proofs, we have factored the system guarantees into a number of *system specifications: Plain Memory*, *Type*, and *Allocator*. The extent of these guarantees differs between low-level and high-level parts of the kernel. For example, the kernel's page-fault handler can access only some parts of the kernel's virtual address space, and it is not allowed to page-fault recursively. We therefore have taken care to allow the specifications to be parameterized with memory regions that can be safely accessed.

The Plain Memory specification models a flat virtual address space in which bytes can be read or written. This specification provides the notion of *blessing* memory regions. It asserts that reading from or writing to a memory region that is read-blessed or write-blessed respectively does not fail. The object-store properties are valid generally only for objects residing in blessed memory. We call instances of this specification a *memory model.*

Normally, these memory models must be implemented in terms of the hardware model's virtual-memory interface.[5] Therefore, each memory model uses one particular page-fault handler.

The Type specification provides operations for converting between typed values and the memory representation of these values as byte sequences, following the ideas described in Section 3.2. There is an instance of Type for each (user-defined or Safe-C++ builtin) data type.

The Allocator specification contains operations for allocating memory blocks in blessed memory. It asserts that within blessed memory regions, each allocated block is accessible at only one virtual address. This property facilitates safe object reads and writes. There are a number of instances of Allocator provided by Safe C++—in particular the static allocator and the stack allocator; for a kernel, there is no predefined heap allocator. However, there can be any number of user-defined allocators written in Safe C++.

[5] However, there are other memory models that are conceivable as well: For example, during the boot process, paging may be turned off, which results in a memory model that operates directly on top of physical memory.

Instantiating the system specifications. For each part of the kernel that is to be verified, we must instantiate the system specifications that are to be used: one memory model and potentially multiple Type and Allocator instances. For the lowest-level parts of the kernel, these instances only include axiomatic knowledge about builtin Safe-C++ types and allocators and about the memory state after boot-up. Higher-level parts can use a richer set of Allocator instances and a more complex memory model that uses a Safe-C++ page-fault handler verified as a lower-level part.

Our memory models are of particular interest because they allow us to use the object-store interface for both low-level and high-level kernel code. In the remainder of this section, we discuss the two memory models we use for these two types of kernel code. In addition, we present another memory model, Physical Memory, which we will use for verifying boot code. We have proven that all of these memory models are indeed instances of Plain Memory.

The "Simple VM" memory model. This memory model is used for verifying low-level kernel code. Its read and write operations are based on our hardware model (Section 4.2). In the Simple VM model, each invocation of the page-fault handler is considered an error. Blessings are based on the contents of the current page table.

Based on the invariant that the kernel's code and static data are always mapped[6] and on the precondition that there is an accessible stack, the Simple VM model can run code that does not rely on page-fault handling and that does not need a custom allocator. We use this model to verify Fiasco's page-table insertion, low-level allocator, and page-fault handler functions.

The "Kernel Memory" memory model. For the bulk of Fiasco kernel code, the Simple VM model does not contain enough features. In particular, it lacks dynamic memory allocation, kernel-virtual memory manipulation, and lazy page-directory updates. Fiasco relies on these features when it dynamically allocates data structures such as thread descriptors from its private memory pool. In this event, it maps new pages into a "master" virtual-address space and lazily updates the kernel regions of user tasks' virtual address spaces from the master copy upon page faults. These lazy updates are completely transparent to the kernel code; for this code, it looks as if the allocated memory "is always there." We reflect this view in our memory model "Kernel Memory."

In this memory model, read and write operations again are based on our hardware model (Section 4.2). The behavior of these operations is similar to the Simple VM model; however, here page-faults invoke the global page-fault handler.

In addition to the Simple VM blessings, the Kernel Memory model also regards as blessed the memory blocks that were allocated using the low-level allocator. Based on this low-level allocator, we can verify a hierarchy of more complex allocators (such as Fiasco's slab allocator).

[6] This invariant needs to be set up by the boot process.

The "Physical Memory" memory model. We have also verified that the our hardware model's physical memory (Section 4.2) is an instance of Plain Memory. In this memory model, read and write operations directly map to the corresponding physical-memory operations; page faults cannot occur. All existing physical memory is blessed.

We will use the Physical Memory model for verifying the part of Fiasco's boot code that runs with paging disabled. This verification will help us establishing the boot-up assumptions of the remaining kernel code.

4.3.2 The object-store layer

The object-store layer is the interface that provides the desired object-store properties. It provides functions for safely manipulating typed objects. This interface is the target language used by our logic compiler.

This layer relies on the guarantees provided by previous section's system specifications. As the object-store layer is independent from the concrete instantiation of these specifications, it works with both the Simple VM model and the Kernel Memory model. Therefore, it is possible to logic-compile *all* kernel code towards the same object-store interface.

We implemented this layer by combining, using some glue logic, the system specifications we described in the previous subsection. The objects reside in a Plain Memory and are accessed using their Plain-Memory addresses. As on a real computer, objects do not have any extra state besides the state stored in Plain Memory. In other words, the object-store operations work on only one state—the memory state.

Based on the system guarantees provided by instances of the system specifications, we were able to prove many object-store properties such as the following:

- Writing to some allocated object does not accidently modify any other allocated object.
- After writing to an allocated object, reading from that object actually returns the value written.
- The order in which you allocate or deallocate objects is irrelevant as long as you deallocate objects with the allocator from which you have allocated it in the first place.

These properties usually take the form of theorem-rewriting rules that allow semiautomatic simplification of and reasoning about state transformers that use only the object-store layer. When reasoning about a sequence of object-store operations, these rewrite rules help by removing uninteresting state modifications.

Consider the following example:

$$\mathsf{addr}_a \neq \mathsf{addr}_b \implies \mathsf{value_of}\big(\mathsf{read}_a(\mathsf{write}_b(state, value))\big) = \mathsf{value_of}(\mathsf{read}_a(state))$$

This rewrite rule states that when reading the value of an object a, it is possible to ignore a preceding write to an object b if a and b have different addresses.

Bootstrapping the verification. It is important to realize that the object-store layer works with each valid instantiation of the system specifications. In particular, it works with both the Simple VM and the Kernel Memory memory models.

Therefore, during the verification we can always use the object-store interface, regardless of whether we are working on low-level parts (such as the page-fault handler) or high-level parts (such as the IPC system). Only the guarantees provided by the object-store layer differ: For the page-fault handler we use the Simple-VM instantiation, thus the object-store properties hold only for objects on the stack and for some statically allocated objects. For the bulk of the kernel code we use the Kernel-Memory instantiation which additionally provides heap allocation and the object-store properties for heap-allocated objects.

Direct hardware access. As we mentioned in Section 4.1, it is sometimes necessary to circumvent the object-store layer to directly access the hardware model, for example to modify the page table, to switch between virtual address spaces, or to access a memory-mapped device register.

This access is easily possible by manipulating the memory state directly using hardware-model functions. Recall that our memory models are actually implemented by providing a functional mapping between the hardware model and the Plain Memory specification. Therefore, the Plain-Memory state *is a* hardware-model state.

Bypassing the object-store layer implies that after a direct hardware-model access it is unknown whether the resulting state still provides the system guarantees. Before the object store can be reasoned about again after such an access, the theorem-prover user needs to reestablished (i. e., prove again) these guarantees explicitly.

4.4 Base Assumptions

Base assumptions are axioms for our verification. We assume them to hold without proving them. The complete list of base assumptions will only be known once we completed the verification. There are two kinds of base assumptions. First, the informal base assumptions that are inherent in our verification approach. Second, the formal base assumptions that will appear as axioms in the Isabelle source code.

Informal base assumptions.

Soundness of Isabelle. We assume that the theorem prover Isabelle does not allow us to infer invalid conclusions. So far Isabelle had only very few soundness bugs.

Correct semantics. We assume that the translation from Safe C++ into HOL correctly captures the semantics of the Safe-C++ source. We try to ensure a correct translation with means from software technology (i. e., performing tests and so on). It is impossible to verify the translation into HOL, because there is no formal semantics of C++.

Compiler and boot loader. We assume a correct compilation of the Fiasco source code. Further, the boot loader should not modify the Fiasco image. With this assumption our results will apply to the running microkernel. In theory it is possible to formalize and verify these assumptions.

Correct hardware model. Our hardware model should be a correct abstraction of an IA32-based system. In theory this assumption could be checked against the VHDL description of an IA32-compliant processor. However, this is illusory not only because of the complexity of these processors.

Formal base assumptions.

Correct booting. The verification starts in a kind of a bootstrap process using the most basic memory model. Therefore we rely on the fact that the boot process sets up a state that fulfills the assumptions of the most basic memory model. As far as possible we will test this boot assumption with an assertion at the end of the booting phase.

Correct stack allocator. We assume that the allocation of automatic variables that is built into the Safe-C++ compiler fulfills the Allocator specification.

We do not plan to make assumptions about the size of the physical memory. Instead we track memory consumption during the verification. This yields theorems that have memory requirements as a precondition in the following form:

> "Property P holds for function f provided that at least s stack space and h heap space is available when calling the function."

5 Conclusion

This paper presents the main ideas for applying source-code verification to the Fiasco microkernel in the VFiasco project. A first challenge of this project is to come up with a semantics of C++ that deals with those features of C++ that are used in the Fiasco sources (including pointer arithmetic and `setjmp/longjmp`). With such a semantics available there is no need to reimplement the kernel in a different programming language. We solve this first challenge by combining state transformers with underspecified functions.

A second challenge in the VFiasco project is to enable high-level reasoning in terms of typed objects during the verification, yet assume only low level hardware properties. Here we use a verification environment that consists of several layers of parametrized specifications.

References

[1] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, Berlin, 1991.

[2] Thierry Cattel. Modelling and verification of a multiprocessor realtime OS kernel. In *7th Internation Conference on Formal Description Techniques*, Berne, Switzerland, October 1994.

[3] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Saski, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[4] G. Duval and J. Julliand. Modeling and verification of the RUBIS μ-kernel with SPIN. In *Proceedings of the First SPIN Workshop*, 1995.

[5] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[7] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, 2001.

[8] D. Evans, J. Guttag, J. Horning, and Y. Tan. LCLint: a Tool for Using Specifications to Check Code. In D. Wile, editor, *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, volume 19:5 of *ACM SIGSOFT Software Engineering Notes*, pages 87–96, New Orleans, USA, December 1994.

[9] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 101–116, New Orleans, Louisiana, February 1999. USENIX Association.

[10] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science 7*. Cambridge University Press, 1988.

[11] J. Harrison. A machine-checked theory of floating point arithmetic. *Lecture Notes in Computer Science*, 1690:113–130, 1999.

[12] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[13] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, 2000.

[14] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. Available at URL http://www.cs.kun.nl/csi/reports/info/CSI-R0007.html, 2000.

[15] B. Jacobs. Exercises in coalgebraic specification. In R. Crole R. Backhouse and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 237–280. Springer, Berlin, 2002.

[16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*, 2001. Available at URL `http://caml.inria.fr/ocaml/`.

[17] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. P. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles (SOSP)*, pages 80–92, Kiawah Island, SC, December 1999.

[18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[19] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, 1996.

[21] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, Berlin, 1994.

[22] W. Reisig. *Elements of Distributed Algorithms*. Springer, Berlin, 1998.

[23] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: A practical tool for OS implementors. In *Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.

[24] J. van den Berg, M. Huisman, B. Jacobs., and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT '99*, number 1827 in LNCS, pages 1–21, 1999.

[25] M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK, November 1998.