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

— Extended Abstract —

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 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 enable high-level reasoning about typed data starting from only low-level knowledge about the hardware and to develop a clean semantics for the subset of C++ used by the kernel. In this extended abstract we present our ideas for tackling the first challenge. We develop a type-safe object store that is based on a hardware model that closely resembles the IA32 virtual-memory architecture, and on guarantees provided by the kernel itself.

1 Introduction

The VFiasco project aims at the mechanical verification of security-relevant properties of the L4-compatible Fiasco microkernel [3].

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 a very expensive process (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 technology. 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 on one aspect of the project: the modeling of a type-safe object store on top of a model of virtual-memory hardware.

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.

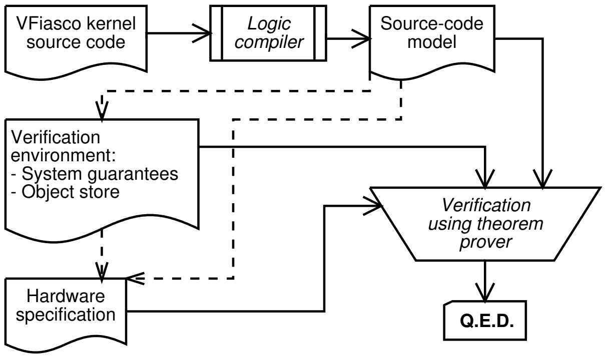

Figure 1: The verification process. Legend: Solid arrows show the flow of data. Dashed arrows indicate a «uses» relationship.

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 [10]. 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 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—a memory that supports reading and writing of typed val-

[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.

ues and that guarantees safe accessibility of these values (as well as other properties). Such an interface 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.

2 Related work

Model checking. Model checking has been successfully applied to several systems in the past [1, 11]. However, this technique can only be applied to abstract models of real systems, because the state space of the real systems is too large. This restriction limits the conclusions that can be drawn from model checking. For instance, in [11] Tullmann and colleagues verify liveness properties of the Fluke kernel's 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 user pointer (which has been abstracted away in the model checker).

Proof-carrying code. Proof-carrying code [9, 8] solves the problem of executing untrusted (user-supplied) code in kernel mode. In this approach, the kernel accepts only those extensions that are accompanied with a valid proof for a given security policy. In a typical application (for example, a network filter) the involved verification is trivial and can be automated.

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. Verification of user-level components is outside the scope of this work; we are undertaking the first step of system verification—proving the kernel correct.

Static source-code checking. There are many tools in the spirit of `lint` that statically analyze source code. For instance the tool presented in [2] has been used to find many bugs in the Linux kernel. Static source-code checking is different from testing in that it analyzes the source code instead of running the system. 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.

Theorem proving. There are a few projects that apply theorem proving at the source code level as we do.

In [7] Liu and colleagues use the theorem prover Nuprl to verify the correctness of network-protocol stacks and to optimize such stacks. There are two major differences to the VFiasco project. First, to enable the verification the original C source code was rewritten in a carefully chosen subset of the functional language Ocaml [6]. 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 do *not* verify the source code. Instead, they verify program transformations.

Our approach to a semantics of C++ is very similar to the one used in the LOOP project for Java [5]. We also use coalgebras to represent statements and expressions. The LOOP project focuses on the verification of Java applications, therefore they can use an object memory that directly represents Java objects [12]. 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 Verification approach

This section sketches the main ideas of our semantics of Safe C++. Please see the full version of this paper for more details, including our approach for dealing with Fiasco's inline-assembler statements [4]. The semantics of Safe C++ is based on two main ideas: *state transformers* and *underspecified functions*.

State transformers. With State transformers (also called *coalgebras*) we adapt the approach of [5] to C++. State transformers allow us to give a relatively simple semantics to statements like `break`, `continue`, and even `goto` and the library routines `setjmp`/`longjmp`. A state transformer is a function (in the mathematical sense) of the following type:

$$\mathsf{St} \longrightarrow \mathtt{ExprResult}\langle\mathsf{St}\rangle$$

Here St stands for the set of all possible states of the system (we elaborate more on structure of St in Section 4). The type $\mathtt{ExprResult}\langle\mathsf{St}\rangle$ is a disjoint (or tagged) union that models the different possible results of C++ expressions (or statements). For instance, if an expression does not terminate, then its result is the distinguished element $\mathsf{Bug} \in \mathtt{ExprResult}\langle\mathsf{St}\rangle$; a `break` statement in a state $s \in \mathsf{St}$ yields $\mathsf{Break}(s)$.

[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.

All C++ expressions and statements (including complex statements) are modeled as state transformers. Composition of state transformers is defined such that the second state transformer is skipped if the first one does not terminate normally. Special statements that regulate flow control are modeled with functions that manipulate state transformers and their results. For instance, loops are wrapped into a function that translate a result of $\mathsf{Break}(s)$ into a normal state, thus resuming execution with the statement that follows the loop.

Underspecified functions. An underspecified function is a function that, although some properties are known, the precise result when applying them is not specified. Thus, in the theorem prover one can only work with the known properties and not with the result of the application. We use underspecified functions to generate the locations of variables and to transform typed values into their byte representation. Underspecified functions allow us to include pointer arithmetic and unsafe type casts in Safe C++.

4 Type safety and 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 we presented in Section 3 to environments with widely differing abstraction levels. For VFiasco our goal is to keep a high-level–language programmer's view during verification while still enabling reasoning about low-level hardware manipulation.

Programmers of high-level languages such as Safe C++, including kernel programmers, make many assumptions about the environment in which their program eventually runs. Table 1 lists a number of such assumptions, which we call *object-store properties.* For example, programmers assume that a program can successfully access typed objects that have been properly allocated.

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—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.

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. Also, we must document all base assumptions[3] we make about the hardware and the Safe-C++ compiler. We describe our hardware model in Section 4.1.

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 4.2 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 the full version of this paper [4].

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 4.2.1 explains how we achieve this goal.

Automation. Based on the object-store properties, we need to provide powerful theorem-rewriting rules that automatically simplify logic-compiled source code without operator intervention as far as possible. We briefly discuss our rewriting rules in Section 4.2.2.

4.1 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.

Currently, the model consists of four main components: the *Physical memory*, the *TLB specification*, *Page-fault handling*, and functions for *reading from and writing to virtual memory*. As the VFiasco project progresses we expect the hardware model to become more detailed, for instance by modeling interrupts and protection levels.

4.2 Verification environment

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

4.2.1 Encapsulating system guarantees

System specifications. We have been able to prove the object-store properties using system properties like those in 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.* The extent of these guarantees differs between low-

[3] In our verification, these base assumptions play the role of axioms.

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
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

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 object-store properties are valid only if the runtime system provides the guarantees in the right column.

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. Here we discuss two of these specifications: *Plain Memory* and *Allocator*.

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 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.[4] Therefore, each memory model uses one particular page-fault handler.

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++.

Instantiating the system specifications. For each part of the kernel that is to be verified, we must instance all system specifications that are to be used: one memory model and potentially multiple Allocator instances. For the lowest-level parts of the kernel, these instances only include axiomatic knowledge about builtin Safe-C++ 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. We have proven that 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 4.1). In this 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[5] 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 4.1). The behav-

[4]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.

[5]This invariant needs to be set up by the boot process. We do not digress into the boot process in this paper.

ior 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).

4.2.2 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. Using this interface, we can verify even low-level Safe-C++ code such as the page-fault handler, which constitutes part of the Kernel Memory model. For this verification, we instantiate the Plain Memory specification using the Simple VM model, which uses only hardware features and does not rely on other Safe-C++ code.

We were able to prove many object-store properties such as "Writing to some allocated object does not accidently modify any other allocated object" and "After writing to an allocated object, reading from that object actually returns the value written." 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.

5 Conclusion

This extended abstract presents the main ideas for applying source-code verification to the Fiasco microkernel in the VFiasco project. The main challenge in this project is to enable high-level reasoning in terms of typed objects during the verification, yet assume only low level hardware properties. We solve this problem with several layers of parametrized specifications.

References

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

[2] 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.

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

[4] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–03–März 2002, Dresden University of Technology, 2002. Available from URL: `http://os.inf.tu-dresden.de/vfiasco/`.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.