The VFiasco approach for a verified operating system

Michael Hohmuth Hendrik Tews

Technische Universität Dresden
Department of Computer Science

vfiasco@os.inf.tu-dresden.de

The quality of today's main-stream operating systems is not sufficient for safety-critical and security-critical applications. In this paper we discuss several possible approaches to build an operating system that is safer and more secure. We especially focus on the approach taken in the VFiasco project on the verification of the Fiasco microkernel operating system. In this project, we use the general-purpose theorem prover PVS to mechanically verify the C++ sources of Fiasco.

1 Introduction

The VFiasco project [8, 14] aims at the formal verification of a small operating-system (OS) kernel, the L4-compatible Fiasco microkernel [6]. In this paper, we explain the reasons that have led us to tackle verifying Fiasco's original C++ source code instead of reimplementing Fiasco in a "safe" programming language such as Haskell or OCaml.

Typical desktop and hand-held computers are used for many functions, often in parallel. These applications frequently include security-sensitive ones, such as online banking, virtual private networks, or digital rights management. This type of computer use imposes two, often conflicting requirements: On the one hand, the mixed-use scenario usually necessitates the use of a full-featured, standard, general-purpose OS. On the other hand, security-sensitive applications must rely on, or *trust,* their operating environment to protect the application's security guarantees.

Standard OSes have become so large that a complete security audit or the formal verification of security properties is absolutely illusory. This fact is illustrated by a steady stream of security-leak disclosures for all major operating systems. It seems that, for the time being, we just have to live with the bugs of standard OSes.

In response, many researchers have tried to reduce the size of a system's trusted computing base by running kernels in untrusted mode in a secure compartment on top of a small security kernel, such as a microkernel or a hypervisor; security-sensitive services run alongside the OS in isolated compartments of their own. This architecture is widely referred to as *kernelized standard OS* or *kernelized system.* In this architecture, the standard OS and applications have been removed from the secure applications' trusted computing base. The root of trust is placed into the small security kernel, which confines unsafe system components in hardware-protected compartments—usually address spaces.

Several recent research projects have shown that contemporary security kernels impose low performance overhead, making them practical. Their source code is several orders of magnitude smaller than that of typical standard-OS kernels, which puts them into the realm of today's formal-verification technology. For example, the Fiasco microkernel has less than 15,000 lines of source code.

The goal of formally verifying a small OS kernel raises the question of how the choice of the kernel's implementation language can facilitate the verification. In discussing this question we focus on a kernel that runs on standard hardware and supports untrusted binary user programs running in separate confined address spaces (i.e., we consider mainstream PCs and hand-helds but exclude, for instance, smart cards).

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) through DFG grant Re 874/2-1.

Modern programming languages with strong type-safety properties can potentially reduce the burden of manually proving a program's safety. However, in this paper we argue that using a "safe" programming language may have quantitative but no qualitative advantages for kernel verification. In other words, with a strong type system one might have fewer proof obligations, but one still has proof obligations of all kinds. In particular, even with the strongest type system there remain type-correctness proof goals that one naively would assume to be subsumed by the type system. Moreover, the proof goals that are indeed subsumed by the type system are simple and can be discharged automatically (when using the right technology).

In the next section we start with the discussion of semi-formal methods to improve the code quality. We then discuss how three different kinds of formal methods can contribute to a secure operating system. Section 3 discusses the approach that we take in the VFiasco project.

2 The need for formal methods

Recently several groups worked on methods to improve the quality of the kernel source code or to counterbalance known weaknesses of the programming language C, which is used for most kernels. We mention only two examples here: With lint-like static checks one can find violations of programming patterns like "validate pointers from user land before dereferencing them" in the source code [4]. To complicate attacks, one can encrypt every pointer in memory such that an attacker cannot walk through and modify kernel data structures in memory [2].

We refer to these and similar methods as *semi-formal*. Although such methods make attacks significantly harder, they cannot be used to guarantee that successful attacks are impossible. Semi-formal methods are a relatively inexpensive means to improve the average level of security. However, they are not sufficient to avert a professional attack.

In contrast, *formal* methods are based on some mathematical theory that can be used to prove certain properties. With a strong type system one can, for instance, use a different type for user-land pointers and thus guarantee that they are never dereferenced without the necessary checks.

In the following we elaborate on different formal methods and their strengths and weaknesses when applied to operating systems. We distinguish three different kinds of formal methods:

- Use a strongly-typed programming language for kernel programming.
- Run the kernel in an environment that enforces security, such as a virtual machine.
- Apply verification and model-checking techniques to the kernel code.

We discuss each of these points in the following subsections.

2.1 "Modern languages" with strong type systems

Several projects develop operating systems in strongly typed functional languages like Haskell or BitC, see for instance [5, 11]. The strong type system makes many typical C-programming errors impossible. Moreover, if the language features abstract types, it can enforce the use of certain interfaces and thereby provide guarantees that certain tests and actions are always applied.

However, there are two problems: First, often some parts of the system are implemented in C or assembly. In House [5], for instance, low-level primitives for allocation and system calls are not written in Haskell. It is clear that in such a case the foreign code might break the type system. Shapiro and colleagues [11] use the specifically designed language BitC to avoid this problem. In BitC one can manipulate address spaces (by writing to the corresponding hardware data structures). However, and this is the second problem, writing a wrong value into a page directory might distort the kernel memory and make all guarantees of the type system meaningless.

On a closer lock it appears that operations like address-space manipulation have a dependent type.[1] Type checking dependent types is undecidable in general. The validity of an address-space manipulation depends on the history of the system, so it is clearly undecidable. We conclude that, even in case the whole system is written in a strongly-typed language, the guarantees of the type system might not be valid because of a wrong address-space manipulation.

In this light, the use of a strong type system appears to be a quantitative rather than a qualitative measure. A strong type system can probably reduce the number of programming errors significantly, however in the presence of address-space manipulations it *cannot* guarantee the type correctness of the program. Thus, even with a strong type system, proof obligations about correct typing remain.

[1] The *type* of some argument depends on the *value* of some other argument. For operating systems the range of valid values sometimes depends on the state of the whole memory.

2.2 Safe environments

One can think of an operating system written in Java and executed by a virtual machine such that important security properties are expressed with Java's type system. This way the virtual machine will detect security violations with its byte-code type checking. However, the security of the whole system depends on the correctness of the virtual machine. Further, to host an operating system one needs operating system functionality itself in the virtual machine. Therefore, virtual machines do not solve the original problem.

With proof-carrying code [10] one can download user-level code into the kernel in a secure way. However, to employ proof-carrying code one has to have a correctly-working minimal kernel with a proof checker in the first place.

2.3 Verification

Software verification establishes properties of a mathematical theory that has been extracted from the software. One can distinguish model checking from verification with denotational, operational or axiomatic semantics. With model checking one explores the finite state space of an abstract model. Subtle errors in parts of the system that have been abstracted away can potentially invalidate the model-checking results.

In an experiment we applied the model checker Spin to Fiasco's code for inter-process communication (IPC) [3]. Although the model contained only a very rudimentary version of Fiasco's IPC (only two threads, no timeouts, no message buffer), the model checker used almost 2 GB main memory and more than 15 GB on the hard disk. We doubt that one can model check the full IPC path, let alone a realistic number of threads with today's hardware.

Up to this point we saw that all discussed techniques can only reduce the number of programming errors and the remaining security risk. None of these techniques can provide guarantees. It remains to look at one approach we have omitted so far: Software verification on the basis of a semantics of the full program. Traditionally, software verification was only applied to theoretically clean, artificial programming languages. Only recently formalisms, logics and tools have been developed to deal, for example, with Java [9, 1]. It has been general belief that it is impossible to verify programs written in C or C++ that exploit features like goto jumps, type casts and `setjmp`/`longjmp`. In the following section we show that a denotational semantics for a subset of C++ that includes these features is surprisingly simple.

3 The VFiasco approach: Verification of unmodified C++ source code

In the preceding section we argued that in order to give guarantees about the functionality of an operating system one finally has to use verification. In the VFiasco project we decided to attempt the verification of the C++ source code of the Fiasco microkernel. The verification of C++ certainly incurs higher verification costs, but has the advantage that we can obtain results for a system that can be used in practice.

Here we sketch how we model two major features of C++: the various jump statements like `break`, `longjmp`, `goto`; and the type cast that can especially be used to turn integers into pointers. From a theoretical point of view one might ask why one should treat these features at all. The point is that in the kernel one *needs* to cast integers into pointers (for the memory management), one *needs* to `longjmp` (to abort an infinite loop of page faults), and one *needs* `goto` (for efficiency and a clearer program structure).

To model the various jumping statements we follow Jacobs' approach in the semantics of Java [9]: We use a complex state that is a disjoint union like $\mathit{ok}(\mathit{mem}) \uplus \mathit{break}(\mathit{mem}) \uplus \mathit{goto}(\mathit{mem} \times \mathit{label}) \uplus \mathit{fail} \ldots$, where ok, break, goto and fail are all injections. This way the complex state captures some kind of execution mode. Apart from the value fail, which models the crash of the program, a complex state contains always the current memory mem. In the following we let the term *state transformer* denote a function that maps a complex state to a complex state. The state transformers form our semantic domain: The semantics of every C++ statement is captured in a state transformer.

State transformers pay attention to the execution mode of their starting state. If the execution mode is ok the next statement is executed as expected and the memory might change. For the other modes the following statements are skipped, except for the case that the current complex state is of the form $\mathit{goto}(m, l)$ and the next statement is labelled with l. In this case the label transforms the complex state into $\mathit{ok}(m)$ and normal execution continues. Similarly to labels, the semantics of loops transforms an end state $\mathit{break}(m)$ into $\mathit{ok}(m)$.

The approach outlined in the preceding paragraph is remarkably simple. It requires neither complete partial orderings nor continuous functions nor continuations. Nevertheless it can handle while loops and all kinds of jump statements, even jumps into and out of nested blocks, such as loops or `then` and `else` clauses. Recently we used this approach to verify

Duff's device, a particularly strange piece of code that uses a `switch` statement to branch into the middle of a `while` loop [13]. This case study showed that, after setting up the right theorem-proving technology (in this case a set of rewrite-lemmas), all those goals that would be subsumed by a stronger type system, were proved automatically. This shows that with the right technology, using a strong type system bears no significant advantage.

As a last point we outline how we model type casts, especially those that cast integers into pointers. There are a few points to note: First, type casts can happen implicitly, for instance, if one writes integers to the memory and reads floats back from the same address. In this case the read operation can cause an error because the bit pattern might not represent a valid floating point number. Second, the cast alone can never cause any errors. An error can only occur if the resulting pointer is used to read data from memory. Third, in general almost all write operations are harmless.[2] One can (possibly partially) overwrite data in the memory but an error can only occur if one attempts to read the original data later.

We exploit the great level of under-specification that is present in the C++ standard to model explicit and implicit type casts [7]. The semantic functions that read from and write to memory are axiomatically specified in a way that leaves many aspects of their behavior open. For instance, we don't specify that integers are stored as two's complement. This way one can derive that reading an integer after writing one on the same address yields the original integer. However, for the case of reading a float on a location where integers have been written, one can derive nothing, not even that the system does not crash. Valid type casts can easily be introduced via additional axioms.

Within our approach, to prove that a program is type correct it is sufficient to prove that it does not crash. Note that our approach does correctly handle the overstressed example of erroneous address-space manipulations. Our approach can handle all data types, as well as compilation environments that guarantee more properties than the C++ standard.

3.1 Dealing with proof obligations for type safety

In this section we shed some more light on how we formalize type correctness and how we deal with it during the verification. For the purpose of this discussion it is enough to consider a memory model like $\mathbb{N} \rightarrow \mathit{Values}$, which is used in the context of one of our lectures [12]. To work with this memory model each variable gets a unique index from $\mathbb{N}$ that determines its slot. The type Values is a disjoint union of all things a variable can hold (i.e., booleans, integers, pointers, ...). This memory model is considerably simpler than what we need for the verification of Fiasco.[3] Note however, that it is dynamically typed, that is, the contents of a variable can change from a boolean to an integer during runtime.

The semantics of variables is captured by functions that read from and write to the memory model. These functions are typed, for instance for integer variables there are two functions $\mathit{write\_int}$ and $\mathit{read\_int}$. The function $\mathit{write\_int}$ writes an integer to a given memory slot, overwriting (and possibly changing the type) of whatever was there before. The function $\mathit{read\_int}$ tests if a given memory slot contains an integer and returns it. If the slot contains no integer $\mathit{read\_int}$ returns the distinguished value *fail*, which signals a program crash.

The approach we just described has the following effect: For every point where a variable is read, we get a proof obligation that requires to show that the accessed slot contains a value of the right type. If one manages to proof that the program does not crash (i.e., that it terminates with something different from *fail*), then one has established that the program is type correct. It is clear that even small programs generate so many type-correctness proof obligations that one must handle them automatically, otherwise even the verification of toy programs would be infeasible.

We use the following approach to handle type-correctness proof obligations.

- Define a type-correctness invariant property that describes the expected type of every relevant memory slot
- Establish simplification rules (called rewrite lemmas in PVS) that automatically discharge the type-correctness proof obligations of $\mathit{read\_int}$ and friends, provided the current memory state fulfils the invariant property
- Establish another set of simplification rules that facilitate the automatic proof that the property from the first point is indeed an invariant for the program under consideration

Our solution is clearly PVS centric, because it is built on *sets of rewrite lemmas*, which provide automatic

[2] The exception are writes that change the address space such that the object code is moved around in virtual memory.

[3] For the verification of C or C++ programs one needs an untyped, byte-wise organized memory [7].

simplification in PVS. However, we believe that our solution can be adopted without problems to other interactive theorem provers that provide automation (for instance to Isabelle and its simplifier). Let us discuss our solution in a bit more detail in the following.

Type correctness invariant. The invariant itself is very simple: It is a predicate on the memory that asserts that certain slots contain values of certain types. Our memory model permits arrays that overlap (with other arrays or with other variables). Therefore the invariant additionally asserts that relevant arrays do not overlap with other arrays or other variables.[4]

To facilitate code reuse and to enable automatic invariant proofs, the invariant predicate is generated inside PVS from an association list that describes the memory slots (i.e., inside the higher-order logic of PVS we have defined a function that maps an association lists to an invariant predicate). Our current code base can handle plain variables and arrays. We are currently working on extensions for records and dynamic structures like linked lists.

Simplification rules for memory accesses. In principle one can directly use the definition of the type-correctness invariant to discharge the type-correctness proof obligations of functions like $read\_int$. However, this does not work automatically in PVS because the invariant is defined with the help of universal quantification and PVS notoriously picks the wrong values during automatic instantiation. Therefore we designed a set of rewrite lemmas that follow trivially from the invariant.

Described in English, the rewrite lemmas look either like "*The invariant implies that the slot of variable v has type t*" or "*The invariant implies that the slot for the i-th element of array a is different from the slot of the j-th element of array b.*" The latter form contains a negated equation. Because of a technical limitation of the rewrite engine of PVS one also needs the form "*The invariant implies … the j-th element of array b is different … the i-th element of array a,*" in which the orientation of the equation is changed.

The number of rewrite lemmas needed grows quadratically with the number of variables. We do not consider this a serious problem because the lemmas and their proofs are highly regular. They can be generated together with the semantics of the program.

With the help of these rewrite lemmas, PVS discharges type-correctness proof obligations automatically. For sequential programs the absence of type errors can be proved automatically: One only has to issue the proof commands to load the rewrite lemmas and to start the simplification process. Type correctness for unbounded while loops requires an invariant for the while loop.

Automatic invariant proofs. The preceding two points about the automation of type-correctness proofs are absolutely essential. We did not tackle any proof without a suitable invariant or suitable rewrite lemmas. To complete the type-correctness proof, one has to show that the predicate from step one is indeed an invariant for the program at consideration. That is, the predicate must be maintained by every state transformer in the semantics of the program. Note that for such an invariant proof the important points are the write operations: Only a write operation can change the type of a slot in the memory and thereby break the invariant.

For our first example verification, a bubble sort algorithm, a direct interactive proof of the invariant property was relatively simple. However, during the verification of Duff's device, it became clear, that for larger sample programs, the invariant proof would became increasingly costly. We therefore decided to set up another set of rewrite lemmas. For for every possible state transformer we designed one lemma that describes how this state transformer maintains a type-correctness invariant. With these rewrite lemmas PVS was able to automatically proof the invariance property.

Our rewrite system for type-correctness invariants works on functions of the semantic domain.[5] Naturally the rewrite system can only handle a small subset of all those functions in the semantic domain that describe type correct programs. However, the computation of the semantics is a highly regular process. Therefore, it is no problem to make the rewrite system general enough such that it can handle all program (fragments) that could also be statically type-checked.

4 Conclusion

In this note we show that to guarantee properties of an operating-system kernel one has to use verification technology. Programming the kernel in a language with a strong type system (in contrast to C or C++) has no

[4]The invariant also states something about the size of arrays in order to treat index-out-of-bounds errors. However, this is not relevant here.

[5]During the computation of the semantics a program is mapped to a certain function (more precisely a state transformer in our approach). The semantic domain is the set of all such functions.

significant advantage because of two reasons. First, the kernel *must* exploit operations that can distort the type safety provided by the language. Therefore even with a strong type system one has to prove type correctness. Second, although the type system subsumes many proof obligations that show up in a verification in a type-unsafe environment, these proof obligation are usually simple and can be solved automatically with the right theorem-proving technology.

Our results apply to all operating systems that provide address-space manipulation. Other OSes, such as library OSes used for embedded microcontrollers, in which all OS and application components are linked together in one address space, can benefit to a larger degree from safe languages because the first reason outlined in the previous paragraph does not apply.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.

[2] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[3] Endrawaty. Verification of the Fiasco IPC implementation. Master's thesis, Technische Universität Dresden, Department of Computer Science, 2005.

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

[5] T. Hallgren. House—Haskell user's operating system and environment. Available at http://www.cse.ogi.edu/~hallgren/House/, April 2005.

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

[7] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. TR No. 187 Inst. für Informatik Universität Freiburg.

[8] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project (extended abstract). In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[9] B. Jacobs and E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security – Theories and Systems (ISSS'03)*, volume 3233 of *Lecture Notes in Computer Science*, pages 134–153. Sprinter, Berlin, 2004.

[10] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., October 1996.

[11] Jonathan Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. Available at www.coyotos.org, June 2005.

[12] H. Tews. Programmverifikation und –spezifikation mit Coalgebren. German lecture script, 2004. Available at wwwtcs.inf.tu-dresden.de/~tews/VeriLecture/.

[13] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for goto and computed jumps. Submitted, 2005. Available via www.vfiasco.org.

[14] H. Tews, H. Härtig, and M. Hohmuth. VFIASCO — towards a provably correct μ–kernel. Technical Report TUD-FI01-1 – January 2001, Technische Universität Dresden, Department of Computer Science, 2001. Available via www.viasco.org.