Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor

Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk

Commercial Formal Methods Team**
Dresden, Germany
uv@lists.askra.de

Abstract. Virtualization engines play a critical role in many modern software products. In an effort to gain definitive confidence on critical components, our company has invested on the formal verification of the NOVA micro hypervisor, following recent advances in similar academic and industrial operating-system verification projects. There are inherent difficulties in applying formal methods to low-level implementations, and even more under specific constraints arising in commercial software development. In order to deal with these, the chosen approach consists in the splitting of the verification effort by combining the definition of an abstract model of NOVA, the verification of fundamental security properties over this model, and testing the conformance of the model w.r.t. the NOVA implementation. This article reports on our experiences in applying formal methods to verify a hypervisor for commercial purposes. It describes the verification approach, and the security properties under consideration, and reports the results obtained.

1 Introduction

Virtualization is prominent in many recent software products. It is used commercially inside cloud services as well as privately for sandboxing or running incompatible legacy applications. Virtualization provides the basis for high-security products that separate applications in disjoint operating-system instances as well as for certain cyber-security products.

The trustworthiness of all these virtualization applications relies fundamentally on the correctness of the hypervisor that implements virtual machine instances on top of the hardware. Encouraged by the success in formal verification applied to large-scale systems in academia [10, 12] and, more recently, also in industrial contexts [17, 6], a number of companies are investing now into formally-verified hypervisors. The authors of this paper worked in a large team together with kernel developers to build a formally verified virtualization solution based on an improved version of the NOVA [23] micro-hypervisor targeting one of the previously mentioned application domains.

** Our company wants to remain anonymous.

A notable case study in formal verification applied to the domain of operating systems is the seL4 project [10]. One could argue that it even constitutes a roadmap or methodology for the verification of low-level large-scale software systems, such as the one we are tackling. However, while the seL4 project was carried out in an academic context, we have to accommodate certain requirements that stem from working in a commercial software development environment.

Challenges. There are some challenges that are specific to the commercial software development context around our targetted hypervisor.

1. The further development of NOVA is driven by feature requests and performance concerns. While the development team is very eager to hear the opinion of the formal methods team, ease of formal verification is not the highest priority when it comes to choice during the development.
2. Release dates are determined according to potential product value and the Company's go-to market strategy. Therefore, it is very likely that the first release will take place before the source code is formally verified. We need to adapt our workflow to these release dates and choose a verification process that permits the release of intermediate results that already provide substantial value to the customer and that can be extended in subsequent releases.
3. We are currently verifying a moving target. Because NOVA lies at the bottom of a stack of components whose design is in constant evolution, the feature set of our version of NOVA changes often and in significant ways. This requires us to adapt the proofs and the correctness and security arguments promptly.
4. NOVA is developed in C++. While there has been work on formalizing aspects of C++ semantics [18, 19]—to the best of our knowledge—there are no mechanized semantics for C++11 as specified by ISO/IEC 14882:2011, which is the flavor pervasively used in the source code.

In order to accommodate to these requirements and restrictions, we have decoupled the high-level properties and their proofs from the low level C++ implementation details. Moreover, we focused on sequential execution NOVA, running on a single core. To this end, we formally prove security properties on an abstract model of the system (written in Coq), and check the correlation between that model and the implementation by model-based testing (which we call conformance testing).

Results. The main objective of our project is to increase the trustworthiness of our virtualization engine using formal methods. In this respect, since the hypervisor is a main building block of the virtualization architecture, the obtained security proofs of the model of the hypervisor are essential to obtain a high-level security property of the whole trusted computing base (TCB).

The number of bugs found and their severeness is considered by the company's management as an important impact indicator of our work. Using our

methodology we have discovered at least a couple of dozen of bugs in the hypervisor component, including a few security-critical bugs, and provided the developers with valuable feedback since the earliest stages of development.

Our methodology also impacts the C++ design quality through all its stages: formal modeling (in cooperation with the C++ developers) drives high quality code reviews in early stages, formal proofs yield the discovery of hard to find corner cases, and the conformance testing provides effective regression testing and excellent test coverage.

Contributions. The main contribution of the paper is to report our experience in applying formal methods in a commercial software-development context. We evaluate advantages and drawbacks of our approach as well as describe our methodology, we discuss possible alternatives and current project status in more technical depth.

Structure of the paper. In Sect. 2 we describe in detail the verification methodology used. In Sect. 3 we provide some background on the NOVA hypervisor. Section 4 describes our Coq formalization. Section 5 presents the high-level security properties that we establish on the model. In Sect. 6 we present our conformance-testing infrastructure. We review related work in Sect. 7 and we present future work and conclusions in Sect. 8.

2 Overview of the Methodology

Developing an abstract model of some real-world system is a common formal verification approach. In our setting, the *real system* is the hypervisor written in C++ and executing in hardware, while the *model* is a formalization within the logic of the Coq proof assistant intended to represent the real system. By their different nature, regardless of the level of detail of the model in question, only empirical evidence can be provided for the adequacy of the representation, and the process of providing this evidence we call *conformance testing*.

There are three main strategies for building the real system and the model:

- Generating the model from the system's source code.
- Generating the system's source code from the model.
- Developing the model and the real system independently.

One benefit of generating the model from the source code, or vice versa, is that one can rely on (or verify) the correctness of the generation mechanism. However, these two strategies pose serious challenges in our setting.

Generating the model from the hypervisor source code is problematic since there is no formalization of the various new features of C++ that are used, and building one is out of scope. Indeed, formal verification is not one of the main objectives of the engineering team developing the C++ implementation of the hypervisor, therefore their design decisions might not always be optimal for verification purposes. Building an ad hoc generator just for our purposes might

be possible, but certainly time consuming, and the output model would be very detailed and thus hard to reason about.

Generating source code from a model leads to similar challenges. One would need to generate source code that executes on bare metal and is aware of special hardware features of various architectures. Moreover this source code has to satisfy also non-verification related objectives like efficiency. Generating such source code from Coq can be too convoluted and, even if we successfully managed to solve these challenges, it would be time consuming to leverage the expert knowledge of the hypervisor developers. Moreover, this approach would also result on a very detailed model.

The chosen approach, an independent development of the model and the real system, provides more flexibility in the design, and the necessary freedom for the C++ source code to address hardware specific issues as well as non verification related objectives, while the model is abstract enough to be easily understandable and easy to reason about. Moreover, this decoupling means that small, low level changes in the C++ source code do not even need to be reflected in the model and thereby grants the model and our proofs much greater stability. However, there is a price to pay: due to the decoupling it is possible that the model and the real system do not agree with each other. In order to overcome this issue, we use model-based testing to provide evidence of the agreement between the implementation and the model, hinting that the properties that we prove in the latter, with a high level of certainty, also hold in the former.

Our starting point is a Coq model of our version of the NOVA hypervisor, based on its design documents. The main components of the Coq model are the hypervisor state and the hypervisor system calls (called *hypercalls* in the rest of the article). The state is an abstraction of the concrete state of the implementation of the hypervisor and comprises all necessary information required to faithfully simulate the behavior of a hypervisor execution. We establish a security property that shows confinement of the resources accessed by a potentially malicious component running on top of the hypervisor on any given execution.

We rely on the Coq extraction mechanism to obtain an executable OCaml version of the model. Around this automatically generated software component, we build a scaffold for running tests in order to empirically assess conformance between the model and the implementation. Additionally, this also constitutes an effective framework to perform fuzzing on the implementation, using our model as an oracle for expected behavior.

3 A Primer on the NOVA Micro Hypervisor

This section provides a brief overview of our improved version of NOVA (referred to simply as hypervisor below) and its high-level design. It also introduces some concepts that are used in the reminder of the paper.

The NOVA micro hypervisor [23] runs directly on the hardware, and it is constructed according to micro-kernel design principles [13] in the tradition of the L4 family [7]. In traditional designs, the *Virtual Machine Monitor* (VMM)

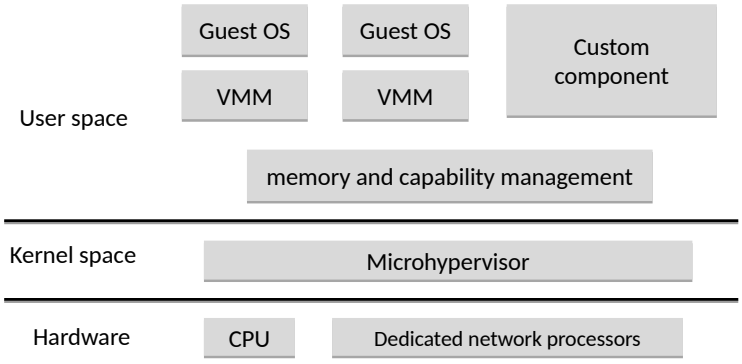

Fig. 1. Potential NOVA based application architecture

is often integrated into the hypervisor for performance reasons. In contrast, in a micro-kernel design, the hypervisor is the sole component running in the most privileged mode of the hardware (host mode ring 0 on x86). The VMM runs as a separate module in unprivileged mode (host mode ring 3 on x86). The hypervisor contains exclusively the functionality that cannot be implemented in unprivileged mode because of performance requirements or hardware restrictions. This design permits to potentially have isolated VMM instances for different guest operating systems as well as to run most device drivers as user applications in unprivileged mode, see Figure 1 for illustration.

Resource separation is an important design objective that the TCB needs to provide. For instance, guest OSs or applications therein must not be able to arbitrarily modify the main memory of other components. Our virtualization architecture relies on the TCB to correctly enforce resource separation, so that potentially malicious guest OSs cannot escape their virtualized environment. The verification of the hypervisor and the correct behavior of its hypercall interface marks our first step towards ensuring the separation property of the whole TCB.

To provide access control, NOVA uses a capability model that is inspired by the take-grant model [14] as well as the EROS capability model [22]. A *capability* is a reference to a resource together with access permissions. NOVA uses three classes of capabilities: memory capabilities (referencing physical memory tiles), object capabilities (referencing *kernel objects*, see below) and I/O capabilities (referencing hardware I/O ports). The access permissions depend on the capability class. For example, permissions in memory capabilities refer directly to the hardware permission bits in the page table entries, while access permissions of kernel objects enable certain hypercalls.

For memory and I/O port capabilities, the capability selectors have a special meaning. For memory, the capability selector denotes the virtual page index at which the referenced memory tile is available in virtual memory. For I/O ports, the selector number is the I/O port number. Therefore, NOVA enforces that I/O port capabilities can only be delegated to identical capability selectors.

Unprivileged programs can reference capabilities via process specific *capability selectors* but cannot directly modify capabilities. The access permissions govern the available operations. For instance, a semaphore capability only permits the down operation on the referenced semaphore if the `dn` permission bit is set. The system might contain several capabilities referencing the same object with different permissions to provide fine-grained access control to different programs. Every capability owner can delegate a capability to a different process if he possesses a capability of the target that permits delegation. Thereby, delegation grants the target process access to the referenced kernel object. The access permissions can be reduced during delegation.

In comparison to other L4 designs, there are a few interesting differences in our version of NOVA. Firstly, delegation is decoupled from inter-process communication. Secondly, there is no recursive capability revocation, one can only delegate empty capabilities to overwrite the contents of certain capability selectors inside a certain process.[1]

3.1 Kernel Objects

The hypervisor provides hypercalls for creation and manipulation of kernel objects. There are five categories of kernel objects.

Processes: processes provide a mechanism for spatial isolation. A process is a collection of capabilities to memory, kernel objects, and I/O ports.[2]

Threads: a thread is a piece of a program that can be independently scheduled. A thread is permanently bound to a process at creation time. Threads can run in *host mode* or *guest mode*. The latter is used to execute a guest OS. Each thread possesses a *user thread-control block* (UTCB) that is used during inter-process communication and which is allocated at thread creation time from the kernel memory pool.

Portals: a portal is a communication endpoint bound to a service-providing thread.

Scheduling objects: scheduling objects provide priorities and execution time. The hypervisor provides a fixed-priority, round-robin scheduler that schedules the threads that possess scheduling objects.

Semaphores: the hypervisor provides counting semaphore objects for thread synchronization.

Kernel objects are allocated in kernel-space memory and are not accessible from unprivilied (user-level) processes (they can only be indirectly referenced through selectors). UTCBs are a special case: for efficiency reasons, they are allocated in kernel memory, but a user-level thread has direct access to its UTCB through a memory selector.

[1] In order to enforce resource revocation from untrusted components in our NOVA version, one needs a trusted component that performs all delegations and tracks them similarly to the mapping database that is part of many L4 implementations.

[2] The NOVA documentation uses *protection domain* instead of *process* and *execution context* instead of *thread* but we stick to traditional terminology here.

3.2 Hypercalls

The hypercalls provide user processes with mechanisms that can be categorized as follows:

Communication: start and terminate inter-process communication calls. Calls always reference a portal and will establish a handshake with the thread that the portal points to. A reply terminates a call and signals the availability of the thread for the next call. For data exchange, the hypervisor appropriately copies the contents of the UTCB from the caller to the callee and back.

Object creation: create kernel objects with a certain set of permissions, and associate them with capability selectors.

Capability delegation: delegation of capabilities to the own or other processes, restricting capability permissions, and deleting capabilities (by delegating empty capabilities).

Object modification: permit modification of relevant aspects of kernel objects (e.g. change the value of a semaphore).

Device management: there is one hypercall to configure direct memory access (DMA) devices and one for associating interrupts to semaphores. Internally both configure the I/O MMU. These device management hypercalls are not relevant for this paper.

There are some interesting aspects of how hardware events are handled in the hypervisor. Firstly, device interrupts are mapped to semaphore-up operations. A thread that wants to wait for an interrupt must perform a down operation on the right semaphore. Secondly, CPU exceptions (e.g., page-fault or divide-by-zero), virtual machine intercepts or exits (VM exits), are mapped to inter-process communication. On behalf of the faulting thread, the hypervisor sets up a call to a portal that depends on the exception or intercept, and fills the UTCB with data describing the exception or intercept as well as the content of the CPU registers of the faulting thread.

4 Coq Model

The Coq abstract model of the hypervisor is essentially defined as a transition system. The states are abstract representations of the hypervisor internal state while the transitions correspond to events performed by (a sequential execution of) the hypervisor. These events can be roughly divided between external events (e.g. a thread issuing a hypercall), and internal events (e.g. the hypervisor resumes a blocked thread).

Structurally, the abstract model is divided in the following components: basic infrastructure, hypervisor state, and semantics. The basic infrastructure component defines the core data structures and lemmas used in the entire development. It contains the definition of the libraries used in the hypervisor semantics, and a large collection of lemmas and tactics for proof automation.

We describe the hypervisor state and semantics in the rest of this section. In Sect. 5 we describe the security properties we prove for this semantics.

4.1 Hypervisor State

The hypervisor state type $\mathcal{K}$ represents the hypervisor internal state. It is defined as a record containing:

- a collection of kernel objects, defined as a partial map from *pointers* (non-negative numbers) to typed kernel objects;
- the addresses of UTCBs to track which parts of the kernel memory might be accesses from unprivileged mode
- architecture-specific state: interrupt mapping, device status, etc;

Kernel objects refer to each other using pointers (e.g. a thread contains a pointer to the process it belongs to). Accessing an object through a pointer may fail if the pointer is not in the partial map, or the mapped object has the wrong type. Therefore, functions to access objects are defined in the error monad (see Sect. 4.2).

Each type of kernel object is defined as a record. Processes are represented as collections of capabilities of a specific type:

- memory capabilities are represented as a map from memory capability selectors (virtual addresses) to physical addresses and permissions;
- object capabilities are represented by a map from object capability selectors to kernel-object pointers;
- I/O capabilities are represented by the set of I/O ports that the process is allowed to access.

Threads contain a stack pointer, a UTCB pointer, a pointer to the associated process, and a status value. The status value is taken from an enumeration type that indicates if the thread is running, available for execution, blocked in a semaphore, etc. The status is not explicitly implemented in the hypervisor, but it is a useful abstraction to have in the model.

Semaphores contain a counter value and a queue of pointers to blocked threads.

Portals and scheduling objects are similarly represented with records, but their contents are not relevant to this paper.

4.2 Semantics

The semantics of the hypervisor is specified as a transition system on the set of kernel states whose transitions are steps that the hypervisor may perform. Steps are divided in categories as follows.

Hypercalls: these are executed by a thread to require a hypervisor service.

Hypervisor events: these are internal to the hypervisor in the sense that they change the state, but are not directly visible for user processes. For example, a semaphore timeout may cause a blocked thread to become unblocked.

Exceptions: this class includes events such as interrupts, DMA access steps, exceptions, etc. Depending on the type of event, they may cause a switch to kernel mode.

Concretely, we define the transition system as a function

$$\mathsf{stepRun} : \mathcal{K} \to \mathcal{S} \to \mathcal{M}(\mathcal{R}, \mathcal{K})$$

where $\mathcal{S}$ is the type representing the hypervisor steps, $\mathcal{R}$ is the result of executing the step, and $\mathcal{M}$ is a non-determinism error monad. This function is extracted to an executable program in OCaml, which we use for conformance testing (see Sect. 6).

The error monad is used to model successful executions as well as failures. Executing a step may fail for several reasons, most typically, when accessing a non-existent object in memory (but we proved an invariant about the absence of certain failures, see Sect. 5 below).

The stepRun function proceeds by first checking *feasibility* of the step to be executed. Feasibility is defined as an over-approximation of the valid steps in a given kernel state. For example, in the case of a hypercall step executed by a thread pointer p, feasibility means that p points to a valid thread object whose status allows execution (i.e. it is not blocked on a semaphore). This notion of feasibility is naturally extended to traces.

If the step is not feasible, execution fails. Otherwise, the function stepRun proceeds to execute the step. Let us illustrate the semantics with the implementation of the create_thread hypercall. We simplify some details that are not relevant for this level of detail. The create_thread hypercall takes four parameters:

$$\mathsf{create\_thread}(\mathit{proc\_sel}, \mathit{th\_sel}, \mathit{utcb\_sel}, \mathit{data})$$

where *proc_sel* is a capability selector referencing the process that shall contain the new thread, *th_sel* is the selector that shall contain the new capability referencing the newly created thread, *utcb_sel* is a memory capability selector describing where the UTCB of the new thread shall be accessible in user virtual memory, and *data* contains other parameters not relevant here (e.g. stack pointer).

We model the create_thread hypercall as a function

$$\mathsf{create\_thread} : \mathcal{K} \to \mathsf{ptr} \to \mathsf{sel} \to \mathsf{sel} \to \mathsf{sel} \to \mathsf{data} \to \mathcal{M}(\mathcal{R}, \mathcal{K})$$

where the first argument is the kernel state where the hypercall is being executed and the second argument is a pointer in the kernel state to the thread executing the hypercall. In Coq, it is defined as follows:

```
create_thread ks t proc_sel th_sel utcb_sel data :=
  p ← get_process ks t;
  if has_ct_perm ks p proc_sel
    then
      ks1, utcb ← allocate_utcb ks proc_sel utcb_sel;
      ks2, th ← new_thread ks1 utcb data;
      ks3 ← map_selector ks2 proc_sel th_sel th;
      return (Success, ks3)
    else
      return (BadPermission, ks)
```

Here we use Coq notations to write monadic-style code: $v \leftarrow f; body$ is a shorthand for $(\lambda v\,.\,body)f$, that is, evaluate *body* with v bound to the result of f. The function proceeds as follows: first, get the process corresponding to the executing thread (t) in the current state (ks). This can fail if t does not point to a valid thread. Then, check that the process referenced by *proc_sel* has permission to create threads. If not, return without modifying the kernel state. Otherwise, allocate a new UTCB (using allocate_utcb), create the new thread object (using new_thread), and finally map a reference to the newly-created thread (using map_selector) at the selector given by the user (*th_sel*).

5 Security Properties

The main security properties we prove for our model are *authority confinement* and *memory confinement*. Authority confinement states that a process cannot gain access to a capability unless it was explicitly delegated to it. In other words, a process cannot "trick" the hypervisor into gaining capabilities by executing a sequence of steps. Memory confinement states that a thread cannot access kernel memory except when it represents a UTCB.

In order to establish these properties on the model, we need to first show a consistency invariant on the semantics. We divide this proof as a conjunction of 10 individual invariants. Most of these invariants refer to internal consistency of our data structures and consistency of the kernel state. For example, memory confinement is proved as an invariant of the state (see below).

Two important examples of invariants proved are *no-dangling pointers* and *semaphore consistency*. No-dangling pointers state that all pointers in a kernel object point to valid objects of the right type. For example, a thread has a valid pointer to its corresponding process; a semaphore's blocked-queue contains pointers to valid threads.

Semaphore consistency refers to the internal consistency of the semaphore structure in a kernel state. It is defined as the conjunction of the following three properties:

- the blocked-queue in any semaphore contains no duplicates and any thread in any semaphore's blocked-queue has a status field indicating it is blocked by a semaphore;
- if a thread status indicates it is blocked by a semaphore, then there exists a semaphore that contains a pointer to the thread in its blocked-queue;
- for any pair of semaphores, their blocked-queues are disjoint.

5.1 Authority Confinement

Consider a partitioning of the processes into two sets that we call *trusted* and *untrusted*. Consider further an initial state k, a kernel event trace $\overline{s}$ and a capability c. Our authority confinement property states that the untrusted processes can never gain access to c as long as the following three conditions are fulfilled.

Firstly, c must not be present in any untrusted process in state k. Secondly, if c is created in $\overline{s}$, it must be created inside a trusted process. Finally, c is never delegated from a trusted to an untrusted process.

This property shows that one can effectively prevent any (untrusted) set of processes $\mathcal{S}$ from gaining access to a certain resource c: one only needs to restrict delegation into $\mathcal{S}$ and the rights of $\mathcal{S}$ to create capabilities by creating new kernel objects. Then, regardless of the actions that are performed inside $\mathcal{S}$, no process inside $\mathcal{S}$ will ever gain access to c.

Authority confinement is proved by a simple induction on the kernel event trace $\overline{s}$, showing that c can only appear inside the untrusted processes if it is either delegated to one of the untrusted processes or created by one of them.

5.2 Memory Confinement

Consider a kernel state k. We say that k satisfies the memory confinement property if for every process p and memory capability m, such that p holds m in k, one of the following holds:

- m does not point in kernel memory, or
- m points to a UTCB.

Memory confinement is proved as an invariant of the semantics. It is an essential security property of the hypervisor: if a process can access kernel memory, then it could potentially access any resource.

6 Conformance Testing

We use conformance testing to provide evidence about the correct implementation of the NOVA hypervisor w.r.t. our abstract model. In turn, this indicates that the properties that we proved for the abstract model hold for running instances of our NOVA version.

For conformance testing we run a kernel event trace (consisting of hypercalls, hypervisor events, and exceptions) both in the hypervisor and in the abstract model, see Figure 2. Running an input trace in the hypervisor or the abstract model produces a final kernel state and an output trace of the kernel events that were actually performed together with hypercall status results. We compare the output traces and the final kernel states and check that the output traces correspond to the input trace. Any mismatch in the comparison indicates a difference in the executions of the abstract model and the hypervisor that needs investigation.

Running a kernel event trace on the hypervisor requires booting the hypervisor together with our test interpreter process, which can execute an arbitrary kernel event trace. The whole testing currently requires certain changes in the hypervisor. They are needed for generating the output trace and the final kernel state. We try to minimize the changes made to the hypervisor in order to ensure that we do not affect the hypervisor semantics.

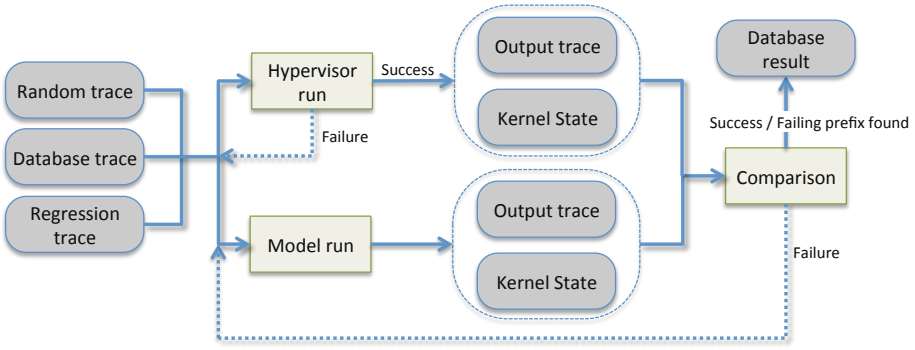

Fig. 2. Scheme of the testing process.

For running the traces in the abstract model, we use the Coq code-extraction facilities to generate OCaml code from the abstract model. We trust the correctness of the Coq extraction mechanisms and assume that the generated code allows evaluating a trace in OCaml according to the Coq model.

Efficiency of the extracted code is an important requirement. For our workload, we found that some data structures in the Coq standard library are not efficient. Concretely, this applies to the standard set library. We proposed a relaxed interface for this library and implemented instances that better fit our workload (see [24]).

Our conformance testing framework provides additional features to ease debugging of failing test cases and simplify our development process. For failing test cases, our framework automatically searches the first step in the input event trace that exhibits a difference in the behavior of the abstract model and the hypervisor. Test data and especially failing test cases can be conveniently investigated via an web front-end.

Traces for running conformance testing come from three different sources: randomly generated, handwritten, and previously-executed traces.

The most important source of traces is our random generator. Simple random trace generation would produce a huge amount of unfeasible steps and almost all hypercalls would fail because of invalid arguments. We therefore use the abstract model to guide the random step generation. Starting from a kernel state, we collect feasible events from the abstract model and randomly chose one of them. For hypercalls we also extract correct arguments and chose with a certain probability only from these arguments. Once a step has been generated, it is run in the abstract model to continue the trace generation with the next kernel state.

The second source of traces for conformance is a set of about 16,000 short handwritten traces that we use for regression testing during the development process. For defining this set, we only require basic coverage of the model and the hypervisor.

Finally, the testing framework supports rerunning traces that were generated in the past. We use this feature for validating whether bugs in the hypervisor or in the model have been fixed.

At the time of writing we have about 12 million executed conformance tests in our data base, of which slightly less than 5% fail for various known bugs in the abstract model or the hypervisor. All these bugs will be addressed in due time before the product release.

7 Related Work

The most relevant to our work is arguably the seL4 project. Initially, Klein et al. established functional correctness of the low-level implementation with respect to a Haskell reference implementation [10]. This correctness proof was extended in several directions, to ensure security properties: integrity [20] and information flow [16]. These properties are proved directly at the implementation level. As we discussed in Sect. 1, we have different challenges: seL4 was developed with the main goal of being verified, whereas our targetted hypervisor is developed as a bedrock for several products in an industrial environment.

The CertiKOS project carried out at Yale University [9] is focused on developing the necessary program logics and infrastructure for the verification of low-level features such as self-modifying code [5] or hardware interrupts and preemptive threads [8], to mention a few. In recent work, Shao [21] proposes redesigning the underlying programming language in which OS kernels are programmed and how it interacts with theorem provers and program logics.

More recently, Liu et al. [15] perform a security analysis of the Goldfish android kernel. In their work, they use the Goanna static analyzer to search for potential vulnerabilities with security implications. They aim at ensuring absence of common coding errors rather than functional correctness.

Our work has strong connections with theorem prover-based testing [4], an instance of model-based testing in which the model is developed in a theorem prover, enabling the proof of properties on top of the model.

Recently, Kosmatov et al. [11] have also combined proofs and testing in the context of hypervisor verification. Concretely, they targeted the virtual memory system of the Axagoros hypervisor. They applied Hoare-style reasoning directly on source code using the Frama-C toolset. When automatic provers fail to discharge proof obligations, they split and isolate the unproven parts and perform all-path testing.

There has also been some work on establishing isolation properties in the context of virtualization [1]. The properties are established in an idealized model with no specific target and therefore, without connection with any particular implementation.

Other work on the verification of large scale systems includes the CompCert C compiler [12] and work carried out by Cousot et al. [2] in the application of static analyses to synchronous control/command in the context of aerospace software. Other recent work targeting this domain includes [3, 25].

Finally, it is worth mentioning that there has been increasing interest in applying formal methods in the high-tech industry: Facebook has been applying static analysis on their mobile applications [6] and Amazon has been using TLA+ to prove properties of concurrent systems at the design level [17].

8 Conclusions

In this paper we have described the challenges we faced when applying formal methods in an industrial context—under a different set of constraints than in most academic work—and the methodology we applied to accommodate to this context. We believe that the lessons we have learned and shared in this paper can be useful when undertaking large-scale verification projects under a similar context.

The work presented in this paper required approximately 3 person-years, which roughly break down into 25% spent in model construction, 35% spent in developing Coq proofs, 15% spent in developing the conformance testing infrastructure (including trace generation), and 25% spent in analyzing conformance testing results.

Throughout the project, approximately a couple of dozen bugs were found in the hypervisor source code. Half of them have been found via code review during model construction and proof. The rest are found via investigation of discrepancies between the model and hypervisor during conformance testing. For most of these bugs, the test cases that trigger them do not crash the hypervisor and do not break any immediate assertion. Therefore, comparing the internal hypervisor state with an expected value (given by the model) is an effective way to show the existence of a bug.

The model is essential during conformance testing as it acts as a executable specification

We should also point out that we found as many bugs in our model (including the Coq model and conformance testing infrastructure), which showed up as false positives during conformance testing.

Testing-related results appeared to be the most efficient way to communicate to the developer teams: we are using metrics like model-based coverage, number of tests and number of reported bugs to convey the impact on increased quality through our work.

The abstract model and its proven security properties demonstrate that there is no security vulnerability in the design of the hypervisor. Millions of conformance tests and the associated coverage provide a convincing argument that design and implementation correlate. Together, our results establish a very high degree of customer confidence in the quality and security of hypervisor-based products.

Future work. There are essentially four dimensions to extend our work. First, we are interested in exploring stronger security properties that can be built on top of our current authority confinement property. Second, we would like to

provide stronger evidence of the connection between the model and the source code. Work is already underway in applying program refinement to construct a chain of increasingly precise models, all the way down to the source code. Third, we aim at extending the verification target to components that run above the hypervisor and that play a crucial role from a security standpoint. In particular, some work has been started on establishing correctness properties at the source level of library code which contains critical data-structures pervasively used in a majority of modules of the system to track notions of ownership and access permission. Finally, the abstract model, the proven properties and conformance testing needs to be extended to parallel execution.

References

1. Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 231–245, 2011.
2. Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.
3. Guillaume Brat, David H. Bushnell, Misty Davies, Dimitra Giannakopoulou, Falk Howar, and Temesghen Kahsai. Verifying the safety of a flight-critical system. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 308–324, 2015.
4. Achim D Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013.
5. Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 66–77, 2007.
6. Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, 2015.
7. Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.
8. Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 170–182, 2008.
9. Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 3, 2011.

10. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220, 2009.
11. Nikolai Kosmatov, Matthieu Lemerre, and Céline Alec. A case study on verification of a cloud hypervisor by proof and structural testing. In *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, pages 158–164, 2014.
12. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
13. J. Liedtke. Toward real μ-kernels. *Communications of the ACM*, 39(9):70–77, September 1996.
14. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, July 1977.
15. Tao Liu and Ralf Huuck. Case study: Static security analysis of the android goldfish kernel. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 589–592, 2015.
16. Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 415–429, 2013.
17. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.
18. Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for c++ multiple inheritance. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 67–80, 2011.
19. Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. A mechanized semantics for C++ object construction and destruction, with applications to resource management. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 521–532, 2012.
20. Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. sel4 enforces integrity. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, pages 325–340, 2011.
21. Zhong Shao. Clean-slate development of certified OS kernels. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 95–96, 2015.
22. Jonathan S. Shapiro and Sam Weber. Verifying the eros confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 166–, Washington, DC, USA, 2000. IEEE Computer Society.
23. Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

24. FireEye Formal Methods Team. Efficiently executable sets used by FireEye. Presented at the 8th Coq Workshop, 2016. Available at `https://github.com/fireeye/MSetsExtra`.
25. Hengjun Zhao, Mengfei Yang, Naijun Zhan, Bin Gu, Liang Zou, and Yao Chen. Formal verification of a descent guidance control program of a lunar lander. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 733–748, 2014.