Scaling symbolic evaluation for automated verification of systems code with Serval

Luke Nelson
University of Washington

James Bornholt
University of Washington

Ronghui Gu
Columbia University

Andrew Baumann
Microsoft Research

Emina Torlak
University of Washington

Xi Wang
University of Washington

Abstract

This paper presents Serval, a framework for developing automated verifiers for systems software. Serval provides an extensible infrastructure for creating verifiers by lifting interpreters under symbolic evaluation, and a systematic approach to identifying and repairing verification performance bottlenecks using symbolic profiling and optimizations.

Using Serval, we build automated verifiers for the RISC-V, x86-32, LLVM, and BPF instruction sets. We report our experience of retrofitting CertiKOS and Komodo, two systems previously verified using Coq and Dafny, respectively, for automated verification using Serval, and discuss trade-offs of different verification methodologies. In addition, we apply Serval to the Keystone security monitor and the BPF compilers in the Linux kernel, and uncover 18 new bugs through verification, all confirmed and fixed by developers.

ACM Reference Format:
Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3341301.3359641

1 Introduction

Formal verification provides a general approach to proving critical properties of systems software [48]. To verify the correctness of a system, developers write a specification of its intended behavior, and construct a machine-checkable proof to show that the implementation satisfies the specification. This process is effective at eliminating entire classes of bugs, ranging from memory-safety vulnerabilities to violations of functional correctness and information-flow policies [49].

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359641

But the benefits of formal verification come at a considerable cost. Writing proofs requires a time investment that is usually measured in person-years, and the size of proofs can be several times or even more than an order of magnitude larger than that of implementation code [49: §7.2].

The *push-button verification* approach [65, 74, 75] frees developers from such proof burden through co-design of systems and verifiers to achieve a high degree of automation, at the cost of generality. This approach asks developers to design interfaces to be *finite* so that the semantics of each interface operation (such as a system call) is expressible as a set of traces of bounded length (i.e., the operation can be implemented without using unbounded loops). Given the problem of verifying a finite implementation against its specification, a domain-specific *automated verifier* reduces this problem to a satisfiability query using symbolic evaluation [32] and discharges the query with a solver such as Z3 [31].

While promising, this co-design approach raises three open questions: How can we write automated verifiers that are easy to audit, optimize, and retarget to new systems? How can we identify and repair verification performance bottlenecks due to path explosion? How can we retrofit systems that were not designed for automated verification? This paper aims to answer these questions.

First, we present Serval, an extensible framework for writing automated verifiers. In prior work on push-button verification [65, 74, 75], a verifier implements symbolic evaluation for *specific* systems, which both requires substantial expertise and makes the resulting verifiers difficult to reuse. In contrast, Serval developers write an *interpreter* for an instruction set using Rosette [80, 81], an extension of the Racket language [35] for symbolic reasoning. Serval leverages Rosette to "lift" an interpreter into a verifier; we use the term "lift" to refer to the process of transforming a regular program to work on symbolic values [76].

Using Serval, we build automated verifiers for RISC-V [85], x86-32, LLVM [54], and Berkeley Packet Filter (BPF) [37]. These verifiers are simple and easy to understand, and inherit from Rosette vital optimizations for free, such as constraint caching [18], partial evaluation [45], and state merging [52]. They are also reusable and interoperable; for instance, we apply the RISC-V verifier to verify the functional correctness of security monitors, and (together with the BPF verifier) to find bugs in the Linux kernel's BPF just-in-time (JIT) compilers.

Second, the complexity of systems software makes automated verification computationally intractable. In practice, this complexity manifests as path explosion during symbolic evaluation, which both slows down the generation of verification queries and leads to queries that are too difficult for the solver to discharge. A key to scalability is thus for verifiers to minimize path explosion, and to produce constraints that are amenable to solving with effective decision procedures and heuristics. Both of these tasks are challenging, as evidenced by the large body of research addressing each [5].

To address this challenge, Serval adopts recent advances in *symbolic profiling* [13], a systematic approach to identifying performance bottlenecks in symbolic evaluation. By manually analyzing profiler output, we develop a catalog of common performance bottlenecks for automated verifiers, which arise from handling indirect branches, memory accesses, trap dispatching, etc. To repair such bottlenecks, Serval introduces a set of *symbolic optimizations*, which enable verifiers to exploit domain knowledge to improve the performance of symbolic evaluation and produce solver-friendly constraints for a class of systems. As we will show, these symbolic optimizations are essential to scale automated verification.

Third, automated verifiers restrict the types of properties and systems that can be verified in exchange for proof automation. To understand what changes are needed to retrofit an existing system to automated verification and the limitations of this methodology, we conduct case studies using two state-of-the-art verified systems: CertiKOS [30], a security monitor that provides strict isolation on x86 and is verified using the Coq interactive theorem prover [79], and Komodo [36], a security monitor that provides software enclaves on ARM and is verified using the Dafny auto-active theorem prover [56]. Our case studies show that the interfaces of such low-level, lightweight security monitors are already finite, making them a viable target for Serval, even though they were not designed for automated verification.

We port both systems to a unified platform on RISC-V. The ported systems run on a 64-bit U54 core [73] on a HiFive Unleashed development board. Like the original efforts, we prove the monitors' functional correctness through refinement [53] and higher-level properties such as noninterference [39]; unlike them, we do so using automated verification of the binary images. We make changes to the original interfaces, implementations, and verification strategies to improve security and achieve proof automation, and summarize the guidelines for making these changes. As with the original systems, the ported monitors do not provide formal guarantees about concurrency or side channels (see §3).

In addition, Serval generalizes to use cases beyond standard refinement-based verification. As case studies, we apply Serval to two existing systems, the Keystone security monitor [55] and the BPF JIT compilers targeting RISC-V and x86-32 in the Linux kernel. We find a total of 18 new bugs, all confirmed and fixed by developers.

Through this paper, we address the key concerns facing developers looking to apply automated verification: the effort required to write verifiers, the difficulty of diagnosing and fixing performance bottlenecks in these verifiers, and the applicability of this approach to existing systems. Serval enables us, with a reasonable effort, to develop multiple verifiers, apply the verifiers to a range of systems, and find previously unknown bugs. As an increasing number of systems are designed with formal verification in mind [29, 55, 71, 88], we hope that our experience and discussion of three representative verification methodologies—CertiKOS, Komodo, and Serval—will help better understand their trade-offs and facilitate a wider adoption of verification in systems software.

In summary, the main contributions of this paper are as follows: (1) the Serval framework for writing reusable and scalable automated verifiers by lifting interpreters and applying symbolic optimizations; (2) automated verifiers for RISC-V, x86-32, LLVM, and BPF built using Serval; and (3) our experience retrofitting two prior security monitors to automated verification and finding bugs in existing systems.

2 Related work

Interactive verification. There is a long and rich history of using interactive theorem provers to verify the correctness of systems software [48]. These provers provide expressive logics for developers to manually construct a correctness proof of an implementation with respect to a specification. A pioneering effort in this area is the KIT kernel [8], which demonstrates the feasibility of verification at the machine-code level. It consists of roughly 300 lines of instructions and is verified using the Boyer-Moore theorem prover [14].

The seL4 project [49, 50] achieved the first functional correctness proof of a general-purpose microkernel. seL4 is written in about 10,000 lines of C and assembly, and verified using the Isabelle/HOL theorem prover [66]. This effort took 11 person-years, with a proof-to-implementation ratio of 20:1. The proof consists of two refinement steps: from an abstract specification to an executable specification, and further to the C implementation. Assembly code (e.g., register save/restore and context switch) and boot code are assumed to be correct and unverified. Extensions include propagating functional correctness from C to machine code through translation validation [72], co-generating code and proof from a high-level language [2, 67], and a proof of noninterference [64].

CertiKOS presents a layered approach for verifying the correctness of an OS kernel with a mix of C and assembly code [40]. CertiKOS adapts CompCert [58], a verified C compiler, to both reason about assembly code directly, as well as prove properties about C code and propagate guarantees to the assembly level. CertiKOS thus verifies the entire kernel, including assembly and boot code, all using the Coq theorem prover [79]. An extension of CertiKOS achieves the first functional correctness proof of a concurrent kernel [41], which is beyond the scope of this paper. We use the publicly available uniprocessor version of CertiKOS, described by Costanzo et al. [30], as a case study for retrofitting systems to automated verification; unless otherwise noted, "CertiKOS" refers to this version. It includes a proof of noninterference.

Despite the manual proof burden, the expressiveness of interactive theorem provers enables developers to verify properties of complex data structures, such as tree sequences in the FSCQ file system [21, 22]. It is also effective in establishing whole-system correctness across layers. Two notable examples are Verisoft [1], which provides formal guarantees for a microkernel from source code down to hardware at the gate level [9]; and Bedrock [24], which verifies web applications for robots down to the assembly level.

Auto-active verification. In contrast to interactive theorem provers, auto-active theorem provers [57] ask developers to write proof annotations [33] on implementation code, such as preconditions, postconditions, and loop invariants. The prover translates the annotated code into a verification condition and invokes a constraint solver to check its validity.

The use of solvers reduces the proof burden for developers, whose task is instead to write effective annotations to guide the proof search. This approach powers a variety of verification efforts for systems software, such as security invariants in ExpressOS [62] and type safety in the Verve OS [87]. Ironclad [43] marks a milestone of verifying a full stack from application code to the underlying OS kernel. The verification takes two steps: first, developers write high-level specifications, implementations, and annotations, all using the Dafny theorem prover [56]; next, an untrusted compiler translates the implementation to a verifiable form of x86 assembly, which the Boogie verifier [6] checks against a low-level specification converted from the high-level Dafny one. The final code runs on Verve. This effort took 3 person-years and achieved a proof-to-implementation ratio of 4.8:1.

Komodo [36] is a verified security monitor for software enclaves. It proves functional correctness and noninterference properties that preclude the OS from affecting or being influenced by enclave behavior. Like Ironclad, the specification and some proofs are written in Dafny. Unlike Ironclad, the implementation is written in a structured form of ARM assembly using Vale [12], which enables Komodo to run on bare hardware. We use Komodo as a case study in this paper.

Push-button verification. The push-button approach considers automated verification as a first-class design goal for systems, redirecting developers' efforts from proofs to interface design, specification, and implementation. To achieve this goal, developers design finite interfaces that can be implemented without using unbounded loops. An automated verifier then performs symbolic evaluation over implementation code to generate constraints and invokes a solver for verification. In contrast to auto-active verification, this approach favors proof automation by limiting the properties and systems that can be verified. Examples of such systems include the Yggdrasil file systems [74], Hyperkernel [65], and extensions to information-flow control using Nickel [75].

These automated verifiers are implemented using Python, with heuristics to avoid path explosion during symbolic evaluation. Inspired by these efforts, Serval builds on the Rosette language [80, 81], which powers a range of symbolic reasoning tools [13: §5]. Rosette provides a formal foundation for Serval, enabling a systematic approach for scaling both the development effort and performance of automated verifiers.

Bug finding. Both symbolic execution [27, 47] and bounded model checking [10] can be used to find bugs in systems code, using tools like KLEE [17], S2E [23], and SAGE [38]; see Baldoni et al. [5] for a survey. These tools are effective at finding bugs, but usually cannot prove their absence.

It is possible to use bug-finding tools to exhaust all execution paths and thus do verification in some settings. Examples include verifying memory safety of a file parser using SAGE [25] and verifying software dataplanes using S2E [34]. Like these tools, Serval uses symbolic evaluation to encode the semantics of implementation code. Unlike them, Serval provides refinement-based verification and more expressive properties (e.g., functional correctness and noninterference).

To improve the performance of symbolic evaluation, research has explored better heuristics for state merging [52] and transformations of implementation code [16, 82]. Serval uses symbolic profiling [13] to identify performance bottlenecks in verifiers and provides symbolic optimizations that exploit domain knowledge to repair these bottlenecks.

3 The Serval methodology

Our goal is to automate the verification of systems code. This section illustrates how Serval helps achieve this goal by providing an approach for building automated verifiers. We start with an overview of the verification workflow (§3.1), followed by an example of how to write, profile, and optimize a verifier for a toy instruction set (§3.2), and how to prove properties (§3.3). Next, we describe Serval's support for verifying systems code (§3.4). We end this section with a discussion of limitations (§3.5).

3.1 Overview

Figure 1 shows the Serval verification stack. With Serval, developers implement a system using standard languages and tools, such as C and gcc. They write a specification to describe the intended behavior of the system in the Rosette language, which provides a decidable fragment of first-order logic: booleans, bitvectors, uninterpreted functions, and quantifiers over finite domains. Serval provides a library to simplify the task of writing specifications, including state-machine refinement and noninterference properties.

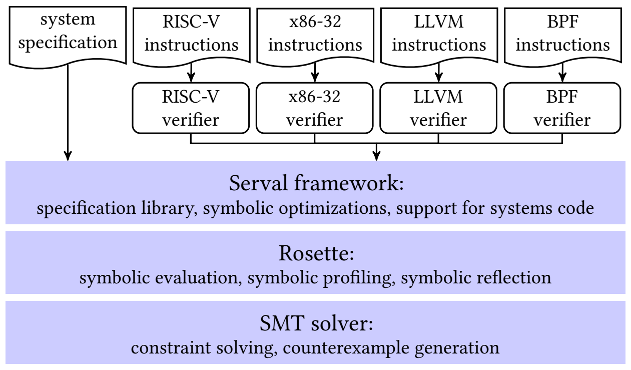


Figure 1. The Serval verification stack. Curved boxes denote verification input and rounded-corner boxes denote verifiers.

An automated verifier, also written in Rosette, reduces the semantics of the implementation code (e.g., in the form of machine instructions) into *symbolic values* through symbolic evaluation. Like previous push-button approaches, this step requires loops to be bounded [65], since otherwise symbolic evaluation diverges. When symbolic evaluation is slow or hangs, system developers invoke the Rosette symbolic profiler [13]. The output identifies the parts of the verifier that cause performance bottlenecks under symbolic evaluation, but it still requires expertise to diagnose the root cause and come up with a fix. Serval provides a set of reusable symbolic optimizations that are sufficient to enable the verifiers from Figure 1 to scale to all the systems studied in this paper. Symbolic optimizations examine the structure of symbolic values (via *symbolic reflection* [81: §2.3]) and use domain knowledge to rewrite them to be more amenable to verification.

Serval employs Rosette to produce SMT constraints from symbolic values (that encode the meaning of specifications or implementations) and invoke a solver to check the satisfiability of these constraints for verification. If verification fails (e.g., due to insufficient specification or incorrect implementation), the solver generates a counterexample, which is visualized by Rosette for debugging.

We emphasize two benefits of the Serval approach. First, it keeps the code and specification cleanly separated, which allows system developers to use standard system languages (C and assembly) and toolchains (gcc and binutils) for implementation; in contrast, other approaches require developers to use specific compilers and languages that are tailored for verification (§6). Second, Serval makes the proof steps fully automatic, requiring no code-level annotations such as loop invariants. This is made possible by requiring systems to have finite interfaces, which enables Serval to generate verification conditions using all-paths symbolic evaluation.

3.2 Growing an automated verifier

Serval comes with automated verifiers for RISC-V, x86-32, LLVM, and BPF. Writing a new verifier in Serval boils down to writing an interpreter and applying symbolic optimizations. As an example, we use a toy instruction set called ToyRISC (simplified from RISC-V), which consists of five instructions (Figure 2). A ToyRISC machine has a program counter pc and two integer registers, a_0 and a_1. For simplicity, it does not have system registers or memory operations. Figure 3 shows a program in ToyRISC, which computes the sign of the value in register a_0 and stores the result back in a_0, using a_1 as a scratch register.

instruction	description	semantics
`ret`	end execution	$pc \leftarrow 0$; halt
`bnez` rs, imm	branch if nonzero	$pc \leftarrow$ if $(rs \neq 0)$ then imm else $pc + 1$
`sgtz` rd, rs	set if positive	$pc \leftarrow pc + 1$; $rd \leftarrow$ if $(rs > 0)$ then 1 else 0
`sltz` rd, rs	set if negative	$pc \leftarrow pc + 1$; $rd \leftarrow$ if $(rs < 0)$ then 1 else 0
`li` rd, imm	load immediate	$pc \leftarrow pc + 1$; $rd \leftarrow imm$

Figure 2. An overview of the ToyRISC instruction set.

```
0: sltz a1, a0   ; a1 <- if (a0 < 0) then 1 else 0
1: bnez a1, 4    ; branch to 4 if a1 is nonzero
2: sgtz a0, a0   ; a0 <- if (a0 > 0) then 1 else 0
3: ret           ; return
4: li   a0, -1   ; a0 <- -1
5: ret           ; return
```

Figure 3. A ToyRISC program for computing the sign of a_0.

Writing an interpreter. Figure 4 lists the full ToyRISC interpreter. It is written in the Rosette language. The syntax is based on S-expressions. For example, `(+ 1 pc)` returns the value of one plus that of `pc`. A function is defined using `(define (name args) body ...)` and the return value is the result of the last expression in the body. For example, the `fetch` function returns the result of `(vector-ref program pc)`. Here `vector-ref` and `vector-set!` are built-in functions for reading and writing the value at a given index of a vector (i.e., a fixed-length array), respectively.

At a high level, this interpreter defines the CPU state as a structure with `pc` and a vector of registers `regs`. The core function `interpret` takes a CPU state and a program, and runs in a fetch-decode-execute loop until it encounters a `ret` instruction. We assume the program is a vector of instructions that take the form of 4-tuples ($opcode$, rd, rs, imm); for instance, "`li a0, -1`" is stored as (`li`, 0, `#f`, −1), where `#f` denotes a "don't-care" value. The `fetch` function retrieves the current instruction at `pc`, and the `execute` function updates the CPU state according to the semantics of that instruction.

Functions provided by Serval are prefixed by "`serval:`" for clarity. The ToyRISC interpreter uses two such functions: `bug-on` specifies conditions under which the behavior is undefined (e.g., `pc` is out of bounds); and `split-pc` concretizes a symbolic `pc` to improve verification performance, which will be detailed later in this section.

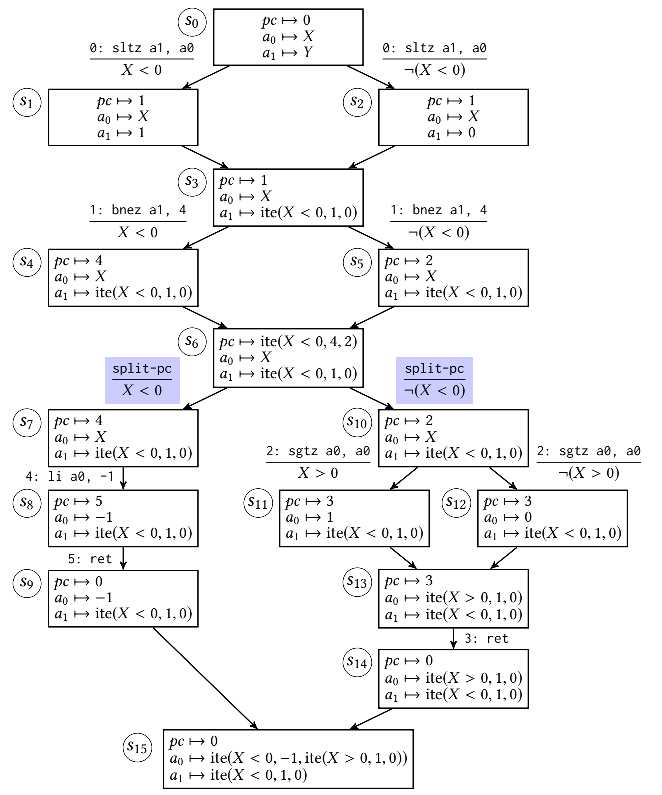
Lifting to a verifier. The interpreter behaves as a regular CPU emulator when it runs with a concrete state. For instance, running it with the code in Figure 3 and $pc = 0, a_0 = 42, a_1 = 0$ results in $pc = 0, a_0 = 1, a_1 = 0$.

```
#lang rosette

; import serval core functions with prefix "serval:"
(require (prefix-in serval: serval/lib/core))

; cpu state: program counter and integer registers
(struct cpu (pc regs) #:mutable)

; interpret a program from a given cpu state
(define (interpret c program)
  (serval:split-pc [cpu pc] c
    ; fetch an instruction to execute
    (define insn (fetch c program))
    ; decode an instruction into (opcode, rd, rs, imm)
    (match insn
      [(list opcode rd rs imm)
        ; execute the instruction
        (execute c opcode rd rs imm)
        ; recursively interpret a program until "ret"
        (when (not (equal? opcode 'ret))
          (interpret c program))])))

; fetch an instruction based on the current pc
(define (fetch c program)
  (define pc (cpu-pc c))
  ; the behavior is undefined if pc is out-of-bounds
  (serval:bug-on (< pc 0))
  (serval:bug-on (>= pc (vector-length program)))
  ; return the instruction at program[pc]
  (vector-ref program pc))

; shortcut for getting the value of register rs
(define (cpu-reg c rs)
  (vector-ref (cpu-regs c) rs))

; shortcut for setting register rd to value v
(define (set-cpu-reg! c rd v)
  (vector-set! (cpu-regs c) rd v))

; execute one instruction
(define (execute c opcode rd rs imm)
  (define pc (cpu-pc c))
  (case opcode
    [(ret)  ; return
      (set-cpu-pc! c 0)]
    [(bnez) ; branch to imm if rs is nonzero
      (if (! (= (cpu-reg c rs) 0))
          (set-cpu-pc! c imm)
          (set-cpu-pc! c (+ 1 pc)))]
    [(sgtz) ; set rd to 1 if rs > 0, 0 otherwise
      (set-cpu-pc! c (+ 1 pc))
      (if (> (cpu-reg c rs) 0)
          (set-cpu-reg! c rd 1)
          (set-cpu-reg! c rd 0))]
    [(sltz) ; set rd to 1 if rs < 0, 0 otherwise
      (set-cpu-pc! c (+ 1 pc))
      (if (< (cpu-reg c rs) 0)
          (set-cpu-reg! c rd 1)
          (set-cpu-reg! c rd 0))]
    [(li)   ; load imm into rd
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]))
```

Figure 4. A ToyRISC interpreter using Serval (in Rosette).

What is more interesting is that given a symbolic state, Rosette runs the interpreter with a ToyRISC program under symbolic evaluation; this encodes *all* possible behaviors of the program, lifting the interpreter to become a verifier. Consider the following code snippet:

```
(define-symbolic X Y integer?)  ; X and Y are symbolic
(define c (cpu 0 (vector X Y))) ; symbolic cpu state
(define program ...)            ; the sign program
(interpret c program)           ; symbolic evaluation
```



s_0 $pc \mapsto 0$ $a_0 \mapsto X$ $a_1 \mapsto Y$
0: sltz a1, a0 $X < 0$
0: sltz a1, a0 $\neg(X < 0)$
s_1 $pc \mapsto 1$ $a_0 \mapsto X$ $a_1 \mapsto 1$
s_2 $pc \mapsto 1$ $a_0 \mapsto X$ $a_1 \mapsto 0$
s_3 $pc \mapsto 1$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
1: bnez a1, 4 $X < 0$
1: bnez a1, 4 $\neg(X < 0)$
s_4 $pc \mapsto 4$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_5 $pc \mapsto 2$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_6 $pc \mapsto \text{ite}(X < 0, 4, 2)$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
split-pc $X < 0$
split-pc $\neg(X < 0)$
s_7 $pc \mapsto 4$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_{10} $pc \mapsto 2$ $a_0 \mapsto X$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
4: li a0, -1
2: sgtz a0, a0 $X > 0$
2: sgtz a0, a0 $\neg(X > 0)$
s_8 $pc \mapsto 5$ $a_0 \mapsto -1$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_{11} $pc \mapsto 3$ $a_0 \mapsto 1$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_{12} $pc \mapsto 3$ $a_0 \mapsto 0$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
5: ret
s_9 $pc \mapsto 0$ $a_0 \mapsto -1$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_{13} $pc \mapsto 3$ $a_0 \mapsto \text{ite}(X > 0, 1, 0)$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
3: ret
s_{14} $pc \mapsto 0$ $a_0 \mapsto \text{ite}(X > 0, 1, 0)$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$
s_{15} $pc \mapsto 0$ $a_0 \mapsto \text{ite}(X < 0, -1, \text{ite}(X > 0, 1, 0))$ $a_1 \mapsto \text{ite}(X < 0, 1, 0)$

Figure 5. Symbolic evaluation of the *sign* program (Figure 3) using the ToyRISC interpreter (Figure 4).

The snippet uses the built-in `define-symbolic` expression to create two symbolic integers X and Y, which represent arbitrary values of type integer. The two symbolic integers are assigned to registers a_0 and a_1, respectively, as part of a symbolic state. Figure 5 shows the process and result of running the interpreter with the symbolic state. Here "*ite*" denotes a symbolic conditional expression; for example, the value of $\text{ite}(X < 0, 1, 0)$ is 1 if $X < 0$ and 0 otherwise.

We give a brief overview of the symbolic evaluation process. Like other symbolic reasoning tools [19], Rosette relies on two basic strategies: symbolic execution [27, 47] and bounded model checking [10]. The former explores each path separately, which creates more opportunities for concrete evaluation but can lead to an exponential number of paths; the latter merges the program state at each control-flow join, which creates compact encodings (polynomial in program size) but can lead to constraints that are difficult to solve [52]. Rosette employs a hybrid strategy [81], which works well in most cases. For instance, after executing `sltz` in Figure 5, Rosette merges the states for the $X < 0$ and $\neg(X < 0)$ cases, resulting in a single state s_3; without merging, it would have to explore twice as many paths.

However, no single evaluation strategy is optimal for all programs. This is a key challenge in scaling symbolic tools [13]. For example, pc in state s_6 becomes symbolic due to state merging of both cases of `bnez`. A symbolic *pc* slows down the verifier—the `fetch` function will explore many infeasible paths—and can even prevent symbolic evaluation from terminating, if the condition at line 20 in Figure 4 becomes symbolic and leads to unbounded recursion. To avoid this issue, the verifier uses the `split-pc` symbolic optimization to force a split on each possible (concrete) *pc* value.

Diagnosing performance bottlenecks. Suppose that the code in Figure 4 did not invoke `split-pc`, causing verification to be slow or even hang. How can we find the performance bottleneck? This is challenging since common profiling metrics such as time or memory consumption cannot identify the root causes of performance problems in symbolic code.

Symbolic profiling [13] addresses this challenge with a performance model for symbolic evaluation. To find the bottleneck in the ToyRISC verifier, we run it with the Rosette symbolic profiler, which produces an interactive web page. The page shows statistics of symbolic evaluation for each function (e.g., the number of symbolic values, path splits, and state merges), and ranks function calls based on a score computed from these statistics to suggest likely bottlenecks.

We find the ranks particularly useful. For example, when profiling the ToyRISC verifier *without* `split-pc`, the top two functions suggested by the profiler are `execute` within `interpret` and `vector-ref` within `fetch`. The first location is not surprising as `execute` implements the core functionality, but `vector-ref` is a red flag. Combined with the statistics showing a large number of state merges in `vector-ref`, one can conclude that this function explodes under symbolic evaluation due to a symbolic *pc*, producing a merged symbolic instruction that represents all possible concrete instructions in the program. This, in turn, causes the verifier to execute every possible concrete instruction at every step.

Symbolic profiling offers a systematic approach for identifying performance bottlenecks during symbolic evaluation. However, symbolic profiling cannot identify performance issues in the solver, which is beyond its scope. One such example is the use of nonlinear arithmetic, which is inherently expensive to solve [46]; one may adopt practice from prior verification efforts to sidestep such issues [36, 43, 65].

Applying symbolic optimizations. Having identified verification performance bottlenecks, where and how should we fix them? Optimizations in the solver are not effective for fixing bottlenecks during symbolic evaluation. More importantly, fixing bottlenecks usually requires domain knowledge not present in Rosette or the solver, such as the set of feasible values for a symbolic *pc*.

Serval provides symbolic optimizations for a verifier to fine-tune symbolic evaluation using domain knowledge. Doing so can both improve the performance of symbolic evaluation and reduce the complexity of symbolic values generated by a verifier; the latter consequently leads to simpler SMT constraints and faster solving.

As for the ToyRISC verifier, state merging on the *pc* slows down symbolic evaluation, while state merging on other registers is useful for compact encodings. Therefore, the verifier applies `split-pc` to the program counter, leaving registers a_0 and a_1 unchanged. After this change, `vector-ref` disappears from the profiler's output. We use this process to identify other common bottlenecks and develop symbolic optimizations (§4).

3.3 Verifying properties

With the ToyRISC verifier, we show examples of properties that can be verified using Serval.

Absence of undefined behavior. As shown in Figure 4, a verifier uses `bug-on` to insert checks based on undefined behavior specified by the instruction set. Serval collects each `bug-on` condition and proves that it must be false under the current path condition [84: §3.2.1]. Serval's LLVM verifier also reuses checks inserted by Clang's UndefinedBehaviorSanitizer [78] to detect undefined behavior in C code.

State-machine refinement. Serval provides a standard definition of state-machine refinement for proving functional correctness of an implementation against a specification [53]. It asks system developers for four specification inputs: (1) a definition of specification state, (2) a functional specification that describes the intended behavior, (3) an abstraction function *AF* that maps an implementation state (e.g., `cpu` in Figure 4) to a specification state, and (4) a representation invariant *RI* over an implementation state that must hold before and after executing a program.

Consider implementation state c and the corresponding specification state s such that $AF(c) = s$. Serval reduces the resulting states of running the implementation from state c and running the functional specification from state s to symbolic values, denoted as $f_{impl}(c)$ and $f_{spec}(s)$, respectively. It checks that the implementation preserves the representation invariant: $RI(c) \Rightarrow RI(f_{impl}(c))$. Refinement is formulated so that the implementation and the specification move in lock-step: $(RI(c) \wedge AF(c) = s) \Rightarrow AF(f_{impl}(c)) = f_{spec}(s)$.

For example, to prove the functional correctness of the *sign* program in Figure 3, one may write a (detailed) specification in Serval as follows:

```
(struct state (a0 a1)) ; specification state

; functional specification for the sign code
(define (spec-sign s)
  (define a0 (state-a0 s))
  (define sign (cond
    [(positive? a0)  1]
    [(negative? a0) -1]
    [else            0]))
  (define scratch (if (negative? a0) 1 0))
  (state sign scratch))

; abstraction function: impl. cpu state to spec. state
(define (AF c)
  (state (cpu-reg c 0) (cpu-reg c 1)))

; representation invariant for impl. cpu state
(define (RI c)
  (= (cpu-pc c) 0))
```

This example shows one possible way to write a functional specification. One may make the specification more abstract, for example, by simply havocing a_1 as a "don't care" value, or by further abstracting away the notion of registers.

Safety properties. As a sanity check on functional specifications, developers should prove key safety properties of those specifications [69]. Safety properties are predicates on specification states. This paper considers two kinds of safety properties: one-safety properties that are predicates on a single specification state, and two-safety properties that are predicates on two specification states [77]. Serval provides definitions of common one- and two-safety properties, such as reference-count consistency [65: §3.3] and noninterference properties [39], respectively.

Take the functional specification of the *sign* program as an example. Suppose one wants to verify that its result depends *only* on register a_0, independent of the initial value in a_1. One may use a standard noninterference property, *step consistency* [70], which asks for an *unwinding relation* ~ over two specification states s_1 and s_2:

```
(define (~ s1 s2)
  (equal? (state-a0 s1) (state-a0 s2))) ; filter out a1
```

Step consistency is formulated as: $s_1 \sim s_2 \Rightarrow \texttt{spec-sign}(s_1) \sim \texttt{spec-sign}(s_2)$. One may write it using Serval as follows:

```
(theorem step-consistency
  (forall ([s1 struct:state]
           [s2 struct:state])
    (=> (~ s1 s2)
        (~ (spec-sign s1) (spec-sign s2)))))
```

Serval's specification library provides the `forall` construct for writing universally quantified formulas over user-defined structures (e.g., `state`).

3.4 Verifying systems code

Having illustrated how to verify a toy program with Serval, we now describe how to extend verification to a real system.

Execution model. Figure 6 shows a typical execution scenario of a system such as an OS kernel or a security monitor. Upon reset, the machine starts in privileged mode, runs boot code (either in ROM or loaded by the bootloader), and ends with an exit to unprivileged mode. From this point, it alternates between running application code in unprivileged mode and running trap handlers in privileged mode (e.g., in response to system calls). Like many previously verified systems [30, 36, 49, 65, 75], we assume that the system runs on a single core, with interrupts disabled in privileged mode; therefore, each trap handler runs in its entirety.

For automated verification, a Serval verifier reduces the system implementation, which consists of trap handlers and boot code, to symbolic values through symbolic evaluation.

For trap handlers, the verifier starts from an architecturally defined state upon entering privileged mode. For instance, it sets the program counter to the value of the trap-vector

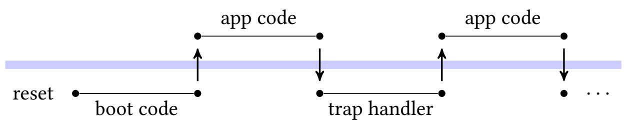


Figure 6. System execution: the lower half and the higher half denote execution in privileged and unprivileged modes, respectively; and arrows denote privilege-level transitions.

register and general-purpose registers to hold arbitrary, symbolic values, as it makes no assumptions about application code. The verifier then performs symbolic evaluation over the implementation until executing a trap-return instruction.

Similarly, for boot code, the verifier starts symbolic evaluation from the reset state as defined by the architecture or bootloader (e.g., the program counter holding a predefined value), and ends upon executing a trap-return instruction.

A verifier may implement a decoder to disassemble instructions from a binary image of the implementation. Serval's RISC-V verifier takes a *validation* approach [83: §5.3]: it invokes `objdump`, a standard tool in binutils, to decode instructions; it also implements an *encoder*, which is generally simpler and easier to audit than a decoder, and validates that the encoded bytes of each decoded instruction matches the original bytes in the binary image. Doing so avoids the need to trust `objdump`, the assembler, or the linker.

Memory model. Serval provides a unified memory model shared by verifiers. The memory model specifies the behavior of common memory access operations, such as load and store. It supports low-level memory operations (e.g., byte-addressing) and is amenable to automated verification.

The Serval memory model borrows ideas from KLEE [17] and CompCert [59]. Like these models, Serval represents memory as a set of disjoint *blocks*, enabling separate reasoning about updates to different blocks. Unlike them, Serval allows system developers to choose an efficient representation for each block, by recursively constructing it using three types of smaller blocks: structured blocks (a collection of blocks of possibly different types), uniform blocks (a sequence of blocks of the same type), and cells (an uninterpreted function over bitvectors); these block types are analogous to structs, arrays, and integer types in C, respectively. For instance, if the implementation accesses a memory region mostly as an array of 64-bit bitvectors or struct types, choosing a block of the same representation reduces the number of generated constraints in common cases, compared to a naïve model that represents memory as a flat array of bytes.

To free system developers from manually choosing the best memory representations, Serval automates the selection. Using `objdump`, it scans the implementation's binary image and extracts top-level memory blocks from symbol tables based on their addresses and sizes; for each top-level block, it produces a memory representation using structured blocks, uniform blocks, and cells, based on types from debugging information. The Serval library performs validity checks on extracted representations (e.g., disjointness and alignment of memory blocks) to avoid the need to trust `objdump`.

3.5 Assumptions and limitations

Serval assumes the correctness of its specification library, the specifications written by system developers, the underlying verification toolchain (Rosette and the solver), and hardware. That said, we discovered two bugs in the U54 RISC-V CPU and implemented workarounds (§6.4).

A verifier written using Serval is a specification of the corresponding instruction set and is trusted to be correct. Since such verifiers are also *executable* interpreters, we write unit tests and reuse existing CPU validation tests [3] to improve our confidence in their correctness. A verifier does not need to trust the development toolchain (gcc and binutils) if it supports verification on binary images, for example, by implementing either a decoder or validation through an encoder (e.g., Serval's RISC-V verifier).

Serval does not support reasoning about concurrent code, as it evaluates each code block in its entirety. We assume the security monitors studied in this paper run on a single core with interrupts disabled while executing monitor code; the original systems made similar assumptions.

The U54 CPU we use is in-order and so not susceptible to recent microarchitectural attacks [15, 51, 61]. But Serval does not support proving the absence of information leakage through such side channels.

Serval explicitly favors proof automation at the cost of generality. It requires systems to be finite (e.g., free of unbounded loops) for symbolic evaluation to terminate; all the systems studied in this paper satisfy this restriction. In addition, it cannot specify properties outside the decidable fragment of first-order logic supported by the specification library (§3.1). The Coq and Dafny theorem provers employ richer logics and can prove properties beyond the expressiveness of Serval, such as noninterference properties using unbounded traces; in §6 we describe such examples and alternative specifications that are expressible in Serval.

4 Scaling automated verifiers

Developing an automated verifier using Serval consists of writing an interpreter to be lifted as a baseline verifier and applying symbolic optimizations to this verifier to fine-tune its symbolic evaluation. Knowing where and what symbolic optimizations to apply is key to verification scalability—none of the refinement proofs of the security monitors terminate without symbolic optimizations (§6.4).

In this section, we summarize common verification performance bottlenecks from our experience with using Serval and how to repair them using symbolic optimizations. To strike a balance between being able to generalize to a class of systems and being effective in exploiting domain knowledge, Serval provides a set of symbolic optimizations as a reusable library that can be tailored for specific systems.

We highlight that symbolic optimizations occur *during* symbolic evaluation, in order to both speed up symbolic evaluation and reduce the complexity of generated symbolic values. Other components in the verification stack perform more generic optimizations: for example, both Rosette and solvers simplify SMT constraints. Symbolic optimizations are able to incorporate domain knowledge by exploiting the structure of symbolic values, as detailed next.

Symbolic program counters. When the program counter becomes symbolic, evaluation of subsequent instruction fetch and execution wastes time exploring infeasible paths, resulting in complex constraints or divergence. Figure 5 shows that state merging can create a symbolic *pc* in the form of conditional *ite* values. The bottleneck is repaired by applying `split-pc` provided by Serval to the *pc*. This symbolic optimization recursively breaks an *ite* value, evaluates each branch separately using a concrete value, and merges the results as more coarse-grained *ite* values; doing so effectively clones the program state for each concrete value, maximizing opportunities for partial evaluation. A computed address (e.g., a function pointer) can also cause the *pc* to become an *ite* value and can be repaired similarly.

It is possible for a symbolic program counter to be wholly unconstrained, for instance, if a system blindly jumps to an address from untrusted sources without checking (i.e., an opaque symbolic value); in this case `split-pc` does not apply and verification diverges. An unconstrained program counter usually indicates a security bug in the system.

Applying `split-pc` before every instruction fetch is sufficient for verifying all the systems studied in this paper. However, choosing an optimal evaluation strategy is challenging in general and may require exploiting domain-specific heuristics [52], for example, by selectively applying `split-pc` to certain program fragments.

Symbolic memory addresses. When a memory address is symbolic (e.g., due to integer-to-pointer conversion or a computed memory offset), it is difficult for a verifier to decide which memory block the address refers to; in such cases, the verifier is forced to consider all possible memory blocks, which can be expensive [17]; some prior verifiers simply disallow integer-to-pointer conversions [65: §3.2].

As an example, suppose the system maintains an array called `procs`; each element is a struct `proc` of C_0 bytes, and a field f resides at offset C_1 of struct `proc`. Given a symbolic *pid* value and a pointer to `procs`$[pid].f$ (i.e., field f of the *pid*-th struct `proc`), a verifier computes an *in-struct offset* to decide what field this pointer refers to, with the following symbolic value: $(C_0 \times pid + C_1) \bmod C_0$. Subsequent memory accesses with this symbolic value cause the verifier to produce constraints whose size is quadratic in the number of fields, as it has to conservatively consider all possible fields in the struct. Note that Rosette does not rewrite this symbolic value to C_1, because the rewrite is unsound due to a possible multiplication overflow.

The root cause of this issue is a *missed concretization* [13]: while procs$[pid].f$ in the source code clearly refers to field f, such information is lost in low-level memory address computation. To reconstruct the information, Serval implements the following symbolic optimization: it matches any symbolic in-struct offset in the form of $(C_0 \times pid + C_1) \bmod C_0$, optimistically rewrites it to C_1, and emits a side condition to check that the two expressions are equivalent *under* the current path condition; if the side condition does not hold, verification fails. Doing so allows the verifier to produce constraints with constant size for memory accesses. Serval implements such optimizations for common forms of symbolic addresses in the memory model (§3.4); all the verifiers inherit these optimizations for free.

Symbolic system registers. Low-level systems manipulate a number of system registers, such as a trap-vector register that holds the entry address of trap handlers and memory protection registers that specify memory access privileges. Verifying such systems requires symbolic evaluation of trap handlers starting from a symbolic state (§3.4), where system registers hold symbolic values. As the specification of system registers is usually complex (e.g., they may be configured in several ways), symbolic evaluation using symbolic values in system registers can explore many infeasible paths and produce complex constraints, leading to poor performance.

However, many system registers are initialized to some value in boot code and never modified afterwards [28], such as a fixed address in the trap-vector register. To speed up verification by exploiting this domain knowledge, Serval reuses the representation invariant, which is written by system developers as part the refinement proof, to rewrite symbolic system registers (or part of them) to take concrete values.

Monolithic dispatching. Trap dispatching is a critical path in OS kernels and security monitors for handling exceptions and system calls. Upon trap entry, dispatching code examines a register that holds the cause (e.g., the system-call number) and invokes a corresponding trap handler. Symbolic evaluation over trap dispatching produces a large, monolithic constraint that encodes all possible trap handlers, since the cause register holds an opaque symbolic value for verification. Proving a property that must hold across trap dispatching is difficult since the solver lacks information to decompose the problem into more manageable parts.

Serval provides `split-cases` to decompose verification properly using domain knowledge. Given a symbolic value x, this optimization asks system developers for a list of concrete values $C_0, C_1, \ldots, C_{n-1}$, such as system-call numbers, and rewrites x to the following equivalent form using *ite* values: $\text{ite}(x = C_0, C_0, \text{ite}(x = C_1, C_1, \ldots \text{ite}(x = C_{n-1}, C_{n-1}, x) \ldots))$.

component (in Rosette)	lines of code
Serval framework	1,244
RISC-V verifier	1,036
x86-32 verifier	856
LLVM verifier	789
BPF verifier	472
total	4,397

Figure 7. Lines counts of the Serval framework and verifiers.

Similarly to `split-pc`, it then proves a property using constraints produced by symbolic evaluation of each branch. Note x appears in the last branch of the generated *ite*, which makes this rewrite sound. But it also means that the optimization leads to effective partial evaluation only when applied to dispatching code that itself consists of a set of paths conditioned on the concrete values $C_0, C_1, \ldots, C_{n-1}$. Applied to such code, this optimization avoids a monolithic constraint and enables reasoning about each trap handler separately.

5 Implementation

Figure 7 shows the code size for the Serval framework and the four verifiers built using Serval, all written in Rosette. The RISC-V verifier implements the RV64I base integer instruction set and two extensions, "M" for integer multiplication and division and "Zicsr" for control and status register (CSR) instructions. The x86-32 verifier models general-purpose registers only and implements a subset of instructions used by the Linux kernel's BPF JIT for x86-32. The LLVM verifier implements the same subset of LLVM as the one in Hyperkernel [65: §3.2]. The BPF verifier implements the extended BPF instruction set [37], with limited support for in-kernel helper functions.

As a comparison, prior push-button LLVM verifiers [65, 75] consist of about 3,000 lines of Python code and lack corresponding optimizations. This shows that verifiers built using Serval are simple to write, understand, and optimize.

6 Retrofitting for automated verification

As case studies on retrofitting systems for automated verification, we port CertiKOS (x86) and Komodo (ARM) to a unified RISC-V platform and make changes to their interfaces, implementations, and verification strategies accordingly; we use superscript "s" to denote our ports, CertiKOS^s and Komodo^s. Like the original efforts, we prove functional correctness and noninterference properties, using Serval's RISC-V verifier on the CertiKOS^s and Komodo^s binary images. This section reports our changes and discusses the trade-offs of the three verification methodologies.

6.1 Protection mechanisms on RISC-V

As shown in Figure 8, both CertiKOS^s and Komodo^s run in machine mode (M-mode), the most privileged mode on

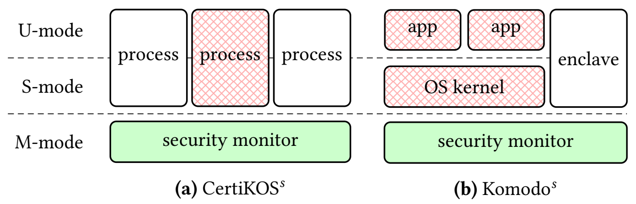


Figure 8. Two security monitors ported to RISC-V (shaded boxes), which aim to protect legitimate components (white boxes) from adversaries (crosshatched boxes).

RISC-V, eliminating the need to further trust lower-level code. Processes or enclaves run in supervisor mode (S-mode), rather than in user mode (U-mode) as in the original systems; doing so enables them to safely handle exceptions [7, 65].

The monitors utilize *physical memory protection* (PMP) and *trap virtual memory* (TVM) on RISC-V [85]. PMP allows M-mode to create a limited number of physical memory regions (up to 8 on a U54 core) and specify access privileges (read, write, and execute) for each region; the CPU performs PMP checks in S- or U-mode. TVM allows M-mode to trap accesses in S-mode to the `satp` register, which holds the address of the page-table root. We will describe how CertiKOS^s and Komodo^s use the two mechanisms later in this section.

For verification, we apply Serval's RISC-V verifier to monitor code running in M-mode, with a specification of PMP and a three-level page walk to model memory accesses in S- or U-mode; we do not explicitly reason about (untrusted) code running in S- or U-mode.

We do not consider direct-memory-access (DMA) attacks in this paper, where adversarial devices can bypass memory protection to access physical memory. RISC-V currently lacks an IOMMU for preventing DMA attacks. We expect protection mechanisms similar to previous work [71] to be sufficient once an IOMMU is available.

6.2 CertiKOS

CertiKOS as described by Costanzo et al. [30] provides strict isolation among multiple processes on x86. It imposes a memory quota on each process, which may consume memory or spawn child processes within its quota. It statically divides the process identifier (PID) space such that each process identified by *pid* owns and allocates child PIDs only from the range $[N \times pid + 1, N \times pid + N]$, where N is configured as the maximum number of children a process can spawn. There is no inter-process communication or resource reclamation. A cooperative scheduler cycles through processes in a round-robin fashion. The monitor interface consists of the following calls:

- `get_quota`: returns current process's memory quota;
- `spawn`(*elf_id*, *quota*): creates a child process using an ELF file identified by *elf_id* and a specified memory *quota*, and returns the PID of the child process; and
- `yield`: switches to the next process.

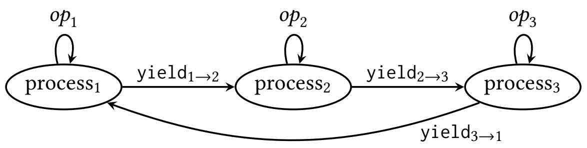


Figure 9. Actions by processes in CertiKOS; op_i denotes either `get_quota`, `spawn`, or a memory access by process i.

Security. CertiKOS formalizes its security using noninterference, specifically, step consistency (§3.3). The intuition is that a process should behave as if it were the only process in the system. To categorize the behavior of processes, we say a *small-step* action to mean either a monitor call or a memory access by a process; and a *big-step* action to mean either a small-step action that does not change the current process, or a sequence consisting of a `yield` from the current process, a number of small-step actions from other processes, and a `yield` back to the original process. Take Figure 9 for example: both "op_1" and "$\texttt{yield}_{1\to2}$; op_2^*; $\texttt{yield}_{2\to3}$; op_3^*; $\texttt{yield}_{3\to1}$" are big-step actions (star indicates zero or more actions).

Using step consistency, CertiKOS proves that the execution of any big-step action by a process should depend only on the portion of the system state *observable* by that process (e.g., the registers and memory it has access to). More formally, we say that two system states s_1 and s_2 are *indistinguishable* to a process p, denoted as $s_1 \sim_p s_2$, to mean that the portions of the states observable to p are the same. Let step(s, a) denote the resulting state of executing action a from state s. Step consistency is formulated so that a process p invoking any big-step action a from two indistinguishable states s_1 and s_2 must result in two indistinguishable states: $s_1 \sim_p s_2 \Rightarrow \text{step}(s_1, a) \sim_p \text{step}(s_2, a)$.

This noninterference specification describes the behavior of each process from the point at which it is spawned. It rules out information flows between any two *existing* processes. However, it does not restrict information flows from a process to its newly created child during `spawn`.

The CertiKOS methodology. CertiKOS takes a modular approach to decompose the system into 32 *layers*: the top layer is an abstract, functional specification of the monitor and the bottom layer is an x86 machine model. Each layer defines an *interface* of operations; except for the bottom layer, a layer also includes an *implementation*, which is a module of the system written in a mixture of C and x86 assembly using the operations from lower layers.

CertiKOS' developers design the layers and split the system implementation into the layers, and write interface specifications and proofs in Coq. Given an implementation in C, they use the `clightgen` tool from the CompCert compiler [58] to translate it into a Coq representation of an abstract syntax tree (AST) in the Clight intermediate language [11]. Each layer proves that the implementation refines the interface.

CertiKOS obtains a proof of functional correctness by composing the refinement proofs across the layers.

CertiKOS proves noninterference over the functional specification at the top layer and augments each layer with a proof that refinement preserves noninterference; doing so propagates the guarantee to the bottom layer. Proving noninterference boils down to proving step consistency for any big-step action. CertiKOS decomposes the proof into proving three separate properties, each about a single, *small-step* action [30: §5]. The three properties are the following, using process_1 in Figure 9 as an example:

- If process_1 performs a small-step action (op_1 or $\texttt{yield}_{1\to 2}$) from two indistinguishable states, the resulting states must be indistinguishable.
- If any process other than process_1 performs a small-step action (op_2, $\texttt{yield}_{2\to 3}$, or op_3), the states before and after the action must be indistinguishable to process_1.
- If process_1 is yielded to ($\texttt{yield}_{3\to 1}$) from two indistinguishable states, the resulting states must be indistinguishable.

By induction, the three properties together imply step consistency for any big-step action of process_1.

CertiKOS uses CompCert to compile the Clight AST to an x86 assembly AST; this process is verified to be correct by CompCert. It then pretty-prints assembly code and invokes the standard assembler and linker to produce the final binary; the pretty printer, assembler, and linker are trusted to be correct. To ensure consistency between the verified code and the proof, CertiKOS' developers delete the original C code and keep the Clight AST in Coq only.

CertiKOS extends CompCert for low-level reasoning; readers may refer to Gu et al. [40] for details. Below we describe two examples. One, CompCert models memory as an infinite number of blocks [59]. This model does not match systems where the memory is limited. CertiKOS alleviates this by using a single memory block of fixed size to represent the entire memory state; this also simplifies reasoning about virtual address translation. Two, CompCert assumes a stack of infinite size, while CertiKOS uses a 4KiB page as the stack for executing monitor calls. To rule out stack overflows, CertiKOS checks the stack usage at the Clight level and augments CompCert to prove that stack usage is preserved during compilation [20].

Retrofitting. The monitor interface of CertiKOS is finite and amenable to automated verification. We make two interface changes to close potential covert channels.

First, CertiKOS' specification of spawn models loading an ELF file as a no-op and does not deduct the memory quota consumed by ELF loading; as a result, the quota in the specification does not match that in the implementation. This is not caught by the refinement proof because CertiKOS trusts ELF loading to be correct and skips the verification of its implementation. One option to fix the mismatch is to explicitly model memory consumed by ELF loading, but this complicates the specification due to the complexity of ELF. CertiKOS^s chooses to remove ELF loading from the monitor so that spawn creates a process with minimum state, similarly to Hyperkernel [65: §4.2]; ELF loading is delegated to an untrusted library in S-mode instead. This also removes the need to trust the unverified ELF loading implementation—any bug in the S-mode library is confined within that process.

Second, spawn in CertiKOS allocates consecutive PIDs: it designates $N \times pid + nr\_children + 1$ as the child PID, where pid identifies the calling process and $nr\_children$ is the number of children it has spawned so far. This scheme mandates that a process disclose its number of children to the child; that is, it allows the child to learn information about an uncooperative parent process through a covert channel. This is permitted by CertiKOS' noninterference specification, which does not restrict parent-to-child flows in spawn. CertiKOS^s eliminates this channel by augmenting spawn with an extra parameter that allows the calling process to choose a child PID; spawn fails if the calling process does not own the requested PID. Note that this change does not allow the calling process to learn any secret about other processes, as the ownership of PIDs is statically determined and public.

For implementation, CertiKOS^s provides the same monitor calls as CertiKOS, except for the above two changes to spawn; a library in S-mode provides compatibility, allowing us to simply recompile existing applications to run on CertiKOS^s. Compared to the original system, there are two main differences in the implementation of CertiKOS^s. First, CertiKOS uses paging for memory isolation. To make it easier to implement ELF loading in S-mode, CertiKOS^s configures the PMP unit to restrict each process to a contiguous region of physical memory and delegates page-table management to S-mode; the size of a process's region equals its quota. The use of PMP instead of paging does not impact the flexibility of the system, as CertiKOS does not support memory reclamation. Second, CertiKOS^s uses a single array of struct proc to represent the process state, whereas CertiKOS splits it into multiple structures across layers.

For verification, Serval proves the functional correctness of CertiKOS^s following the steps outlined in §3.4. As for noninterference, we cannot express CertiKOS' specification in Serval, because it uses a big-step action, which can contain an *unbounded* number of small-step actions (§3.5). Instead, we prove two alternative noninterference specifications that are expressible in Serval. First, as mentioned earlier, CertiKOS develops three properties that together imply noninterference; each property reasons about a small-step action. We reuse and prove these properties as the noninterference specification for CertiKOS^s. In addition, we prove the noninterference specification developed in Nickel [75]; it is also formulated as a set of properties, each using a small-step action. This specification supports a more general form of noninterference called *intransitive* noninterference [64, 70], which enabled us to catch the PID covert channel in spawn.

6.3 Komodo

Komodo [36] is a software reference monitor that implements an SGX-like enclave mechanism on an ARM TrustZone platform in verified assembly code. The Komodo monitor runs in TrustZone's *secure world*, and is thus isolated from an untrusted OS. It manages *secure pages*, which can be used only by enclaves; and *insecure pages*, which can be shared by the OS and enclaves. Komodo provides two sets of monitor calls: one used by the OS to construct, control the execution of, and communicate with enclaves; and the other used by an enclave to perform remote attestation, dynamically manage enclave memory, and communicate with the OS.

Below we briefly describe the lifetime of an enclave in Komodo. For construction, the OS first creates an empty enclave and page-table root using `InitAddrspace`. It adds to the enclave a number of idle threads using `InitThread`, 2nd-level page tables using `InitL2PTable`, and secure and insecure data pages using `MapSecure` and `MapInsecure`, respectively. Upon completion, the OS switches to enclave execution by invoking either `Enter` to run an idle thread or `Resume` to continue a suspended thread. Control returns to the OS upon an enclave thread either invoking `Exit` or being suspended (e.g., due to an interrupt). For reclamation, the OS invokes `Stop` to prevent further execution in an enclave and `Remove` to free its pages.

Security. Like CertiKOS (§6.2), Komodo specifies noninterference as step consistency with big-step actions. A big-step action in Komodo both starts and ends with the OS. For example, both "op_{OS}" and "$\texttt{Enter}_{\text{OS}\to 1}; op_1^*; \texttt{Exit}_{1\to\text{OS}}$" in Figure 10 are big-step actions. Unlike CertiKOS, Komodo considers two types of *observers*: (1) an enclave and (2) an adversary consisting of the OS with a colluding enclave. The intuition is that an adversary can neither influence nor infer secrets about any big-step action. Formally, noninterference is formulated so that any big-step action a from two indistinguishable states s_1 and s_2 to an observer L must result in two indistinguishable states to L: $s_1 \sim_L s_2 \Rightarrow \text{step}(s_1, a) \sim_L \text{step}(s_2, a)$.

A subtlety is that Komodo permits legitimate information flows (e.g., intentional declassification [60]) between an enclave and the OS, which violates step consistency. For example, an enclave can return an exit value, which declassifies part of its data. Komodo addresses this issue by relaxing its noninterference specification with additional axioms.

The Komodo methodology. Komodo's developers first built an unverified prototype of the monitor in C, before using a combination of the Dafny [56] and Vale [12] languages to specify, implement, and verify the monitor. Specifications are written in Dafny, including a formal model of a subset of the ARMv7 instruction set, the functional specification of the security monitor, and the specification of noninterference. The monitor's implementation is written in Vale, and consists of structured assembly code together with proof annotations, such as pre- and post-conditions and invariants.

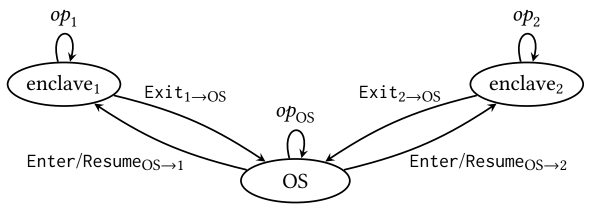

op_1
enclave$_1$
$\texttt{Exit}_{1\to\text{OS}}$
op_{OS}
$\texttt{Exit}_{2\to\text{OS}}$
op_2
enclave$_2$
$\texttt{Enter/Resume}_{\text{OS}\to 1}$
OS
$\texttt{Enter/Resume}_{\text{OS}\to 2}$

Figure 10. Actions by enclaves and the OS in Komodo; op_i and op_{OS} denote either monitor calls or memory accesses by enclave i and by the OS, respectively.

To simplify proofs about control flow, the ARM specification does not explicitly model the program counter; there are no jump or branch instructions. Instead, it models structured control flow: if-statements, while-loops, and procedure calls that correspond to language features in Vale.

The Vale tool is not trusted. It emits a Dafny representation of the program, along with a purported proof (generated from the annotations) of its correctness, which are checked by the Dafny theorem prover. A small (trusted) Dafny program pretty-prints the assembly code for the verified monitor by emitting the appropriate labels and jump instructions, and inlining subprocedure bodies at call sites.

The Komodo implementation uses Vale similarly to a macro assembler with proof annotations. Each procedure consists of a small number of instructions (typically 10 or fewer, to maintain acceptable verification performance), with explicit Hoare-style pre- and post-conditions. Procedures take explicit arguments referring to concrete operands (machine registers or compile-time constants) and abstract values ("ghost variables") that are used in proof annotations but do not appear in the final program.

Register allocation is managed by hand. Low-level procedures with many call-sites (e.g., the procedure that updates a page-table entry) take their parameters in arbitrary registers with explicit preconditions that require the registers to be disjoint. Higher-level procedures (e.g., portions of the system call handlers) use explicit hard-coded register allocations. This makes code reuse challenging, but improves verification times, since the verifier need not consider the alternatives. In practice, it is manageable for a RISC architecture with a large register set, but cumbersome.

Retrofitting. Komodo^s is a RISC-V port of the unverified C prototype of Komodo. We choose the C prototype over the Vale implementation because it is easier to reuse and understand. The C prototype lacks attestation and SGX2-like dynamic memory management [63], and so does Komodo^s.

The Komodo interface is finite and amenable to automated verification. We make two changes to account for architectural differences between ARM and RISC-V. First, because RISC-V has three-level paging (unlike ARM, which has two levels), we add a new `InitL3PTable` call for allocating a 3rd-level page table. Second, we change the page mapping calls, `MapSecure` and `MapInsecure`, to take the physical page number of a 3rd-level page table and a page-table entry index, instead of a virtual address as in Komodo. This change avoids increasing the complexity of page walks in the monitor due to three-level paging, and is similar to seL4 [49], Hyperkernel [65], and other page-table management calls in Komodo.

For implementation, Komodo^s combines PMP, paging, and TVM to achieve memory isolation, as follows. First, it configures the PMP unit to prevent the OS from accessing secure pages. Second, the interface of Komodo^s controls the construction of enclaves' page tables, similarly to that of Komodo; this prevents an enclave from accessing other enclaves' secure pages or its own page-table pages. Third, Komodo^s executes enclaves in S-mode; it uses TVM to prevent enclaves from modifying the page-table root (§6.1).

Komodo^s reuses Komodo's architecture-independent code and data structures. We additionally modify Komodo^s by replacing pointers with indices in struct fields. This is not necessary for verification, but simplifies the task of specifying representation invariants for refinement (§3.3). For instance, it uses a page index rather than a pointer to the page; this avoids specifying that the pointer is page-aligned.

Verifying the functional correctness of Komodo^s is similar to verifying that of CertiKOS^s. For noninterference, we cannot express Komodo's specification in Serval due to the use of big-step actions. Since Komodo does not provide properties using small-step actions as in CertiKOS, we prove Nickel's specification instead [75]. However, it is difficult to directly compare the two noninterference specifications, especially when they are written in different logics and tools. We construct *litmus tests* to informally understand their guarantees. For example, both specifications preclude the OS from learning anything about the contents of memory belonging to a finalized enclave. However, the noninterference specification of Komodo permits, for instance, the specification of a monitor call that overwrites enclave memory with zeros, while that of Komodo^s precludes it. Note that such bugs are prevented in Komodo's functional specification. There may also exist bugs precluded by the noninterference specification of Komodo but not by that of Komodo^s.

6.4 Results

Figure 11 summarizes the sizes of CertiKOS^s and Komodo^s, including both the implementations (in C and assembly) and the specifications (in Rosette); and verification times using the RISC-V verifier on an Intel Core i7-7700K CPU at 4.5 GHz, broken down by theorem and gcc's optimization level for compiling the implementations.

Porting and verifying the two systems using Serval took roughly four person-weeks each. The time is reduced by the fact that we benefit from being able to reuse the original systems' designs, implementations, and specifications. With automated verification, we focused our efforts on developing specifications and symbolic optimizations, as follows.

	CertiKOS^s	Komodo^s
lines of code:		
implementation	1,988	2,310
abs. function + rep. invariant	438	439
functional specification	124	445
safety properties	297	578
verification time (in seconds):		
refinement proof (`-O0`)	92	275
refinement proof (`-O1`)	138	309
refinement proof (`-O2`)	133	289
safety proof	33	477

Figure 11. Sizes and verification times of the monitors.

It is difficult to write a specification for an entire system at once. We therefore take an incremental approach, using LLVM as an intermediate step. First, we compile the core subset of a monitor (trap handlers written in C) to LLVM, ignoring assembly and boot code. We write a specification for this subset and prove refinement using the LLVM verifier; this is similar to prior push-button verification [65, 75]. Next, we reuse and augment the specification from the previous step, and prove refinement for the binary image produced by gcc and binutils, using the RISC-V verifier. This covers all the instructions, including assembly and boot code, and does not depend on the LLVM verifier. Last, we write and prove safety properties over the (augmented) specification. In our experience, the use of LLVM adds little verification cost and makes the specification task more manageable; it is also easier to debug using the LLVM representation, which is more structured than RISC-V instructions.

An SMT solver generates a counterexample when verification fails, which is helpful for debugging specifications and implementations. But the solver can be overwhelmed, especially when a specification uses quantifiers. To speed up counterexample generation, we adopt the practice from Hyperkernel of temporarily decreasing system parameters (e.g., the maximum number of pages) for debugging [65: §6.2].

Symbolic optimizations are essential for the verification of the two systems. Disabling symbolic optimizations in the RISC-V verifier causes the refinement proof to time out (after two hours) for either system under any optimization level, as symbolic evaluation fails to terminate. The verification time of the safety proofs is not affected, as the proofs are over the specifications and do not use the RISC-V verifier.

We first developed all the symbolic optimizations in the RISC-V verifier during the verification of CertiKOS^s, using symbolic profiling as described in §3.2; these symbolic optimizations were sufficient to verify Komodo^s. However, verifying a Komodo^s binary compiled with `-O1` or `-O2` took five times as much time compared to verifying one compiled with `-O0`; it is known that compiler optimizations can increase the verification time on binaries [72]. To improve this, we continued to develop symbolic optimizations for Komodo^s. Specifically, one new optimization sufficed to reduce the verification time of Komodo^s for `-O1` or `-O2` to be close to that for `-O0` (it did not impact the verification time of other systems). Finding the root cause of the bottleneck and developing the symbolic optimization took one author less than one day. This shows that symbolic optimizations can generalize to a class of systems, and that they can make automated verification less sensitive to gcc's optimizations.

As mentioned in §3.5, while developing Serval, we wrote new interpreter tests and reused existing ones, such as the `riscv-tests` for RISC-V processors. We applied these tests to verification tools, and found two bugs in the QEMU emulator, and one bug in the RISC-V specification developed by the Sail project [4], all confirmed and fixed by developers. We also found two (confirmed) bugs in the U54 core: the PMP checking was too strict, improperly composing with superpages; and performance-counter control was ignored, allowing any privilege level to read performance counters, which creates covert channels. To work around these bugs, we modified the implementation to not use superpages, and to save and restore all performance counters during context switching.

6.5 Discussion

Specification and verification. As detailed in this section, CertiKOS uses Coq and Komodo uses Dafny. Both theorem provers provide richer logics than Serval and can express properties that Serval cannot, as well as reason about code with unbounded loops. This expressiveness comes at a cost: Coq proofs impose a high manual burden (e.g., CertiKOS studied in this paper consists of roughly 200,000 lines of specification and proof), and Dafny proofs involve verification performance problems that can be difficult to debug [36: §9] and repair (e.g., requiring use of triggers [42: §6]). Building on Rosette, Serval chooses to limit system specifications to a decidable fragment of first-order logic (§3.1) and implementations to bounded code. This enables a high degree of proof automation and a systematic approach to diagnosing verification performance issues through symbolic profiling [13].

Regardless of methodology, central to verifying systems software is choosing a specification with desired properties. Our case studies involve three noninterference specifications. What kinds of bugs can each specification prevent? While we give a few examples in §6.2 and §6.3, we have no simple answer. We would like to explore further on how to contrast such specifications and which to choose for future projects.

Implementation. CertiKOS requires developers to decompose a system implementation, written in a mixture of C and assembly, into multiple layers for verification. For instance, instead of using a single struct `proc` to represent the process state, it splits the state into various fine-grained structures, each with a small number of fields. Designing such layers requires expertise. Komodo requires developers to implement a system in structured assembly using Vale, which restricts the type of assembly that can be used (e.g., no support for unstructured control flow or function calls). This also means that it is difficult to write an implementation in C and reuse the assembly code produced by gcc. Serval separates the process of implementing a system from that of verification, making it easier to develop and maintain the implementation. Developers write an implementation in standard languages such as C and assembly. But to be verifiable with Serval, the implementation must be free of unbounded loops.

Both CertiKOS and Komodo require the use of verification-specific toolchains for development. For instance, CertiKOS depends on the CompCert C compiler, and Komodo uses Vale to produce the final assembly. Serval's verifiers can work on binary images, which allows developers to use standard toolchains such as gcc and binutils.

7 Finding bugs via verification

Besides proving refinement and noninterference properties, we also apply Serval to write and prove *partial* specifications [44] for systems. These specifications do not capture full functional correctness, but provide effective means for rapidly exploring potential interface designs and exposing subtle bugs in complex implementations.

Keystone. We applied Serval to analyze the interface design of Keystone [55], an open-source security monitor that implements software enclaves on RISC-V. Keystone uses a dedicated PMP region for each enclave to provide memory protection, rather than using paging as in Komodo (§6.3). Since Keystone was in active development and did not have a formal specification, we wrote a functional specification based on our understanding of its design. As a sanity check, we wrote and proved safety properties over the specification. We manually compared our specification with Keystone's implementation, and found the following two differences.

First, Keystone allowed an enclave to create more enclaves within itself, whereas our specification precludes this behavior. Allowing an enclave to create enclaves violates the safety property that an enclave's state should not be influenced by other enclaves, which we proved over our specification using Serval. Second, Keystone required the OS to create a page table for each enclave and performed checks that the page table was well-formed; our specification does not have this check, as PMP alone is sufficient to guarantee isolation for enclaves. Based on the analysis, we made two suggestions to Keystone's developers: disallowing the creation of enclaves inside enclaves and removing the check on page tables from the monitor; both have been incorporated into Keystone.

We also ran the Serval LLVM verifier on the Keystone implementation and found two undefined-behavior bugs, oversized shifting and buffer overflow, both on the paths of three monitor calls. We reported these bugs, which have been fixed by Keystone's developers since.

BPF. The Linux kernel allows user space to extend the kernel's functionality by downloading a program into the kernel, using the extended BPF, or BPF for short [37]. To improve performance, the kernel provides JIT compilers to translate a BPF program to machine instructions for native execution. For simplicity, a JIT compiler translates one BPF instruction at a time. Any bugs in BPF JIT compilers can compromise the security of the entire system [83].

Using Serval, we wrote a checker for BPF JIT compilers, by combining the RISC-V, x86-32, and BPF verifiers. The checker verifies a simple property: starting from a BPF state and an equivalent machine state (e.g., RISC-V), the result of executing a single BPF instruction on the BPF state should be equivalent to the machine state resulting from executing the machine instructions produced by the JIT for that BPF instruction. The checker takes a JIT compiler written in Rosette, invokes the BPF verifier and a verifier for a target instruction set (e.g., RISC-V) to verify this property, and reports violations as bugs. As the JIT compilers in the Linux kernel are written in C, we manually translated them into Rosette. Currently, the translation covers the code for compiling BPF arithmetic and logic instructions; this process is syntactic and we expect to automate it in the future.

Using the checker, we found a total of 15 bugs in the Linux JIT implementations: 9 for RISC-V and 6 for x86-32. These bugs are caused by emitting incorrect instructions for handling zero extensions or bit shifts. The Linux kernel has accumulated an extensive BPF test suite over the years, but it failed to catch the corner cases found by Serval; this shows the effectiveness of verification for finding bugs. We submitted patches that fix the bugs and include additional tests to cover the corresponding corner cases, based on counterexamples produced by verification. These patches have been accepted into the Linux kernel.

8 Reflections

The motivation for developing Serval stems from an earlier attempt to extend push-button verifiers to security monitors. After spending one year experimenting with this approach, we decided to switch to using Rosette, for the following reasons. First, the prior verifiers support LLVM only and cannot verify assembly code (e.g., register save/restore and context switch), which is critical to the correctness of security monitors. Extending verification to support machine instructions is thus necessary to reason about such low-level systems. In addition, the verifiers encode the LLVM semantics by directly generating SMT constraints rather than lifting an easy-to-understand interpreter to a verifier via symbolic evaluation; the former approach makes it difficult to reuse, optimize, and add support for new instruction sets. On the other hand, Rosette provides Serval with symbolic evaluation, partial evaluation, the ability to lift interpreters, and a symbolic profiler. Rosette's symbolic reflection mechanism, originally designed for lifting Racket libraries [81: §2.3], is a good match for implementing symbolic optimizations.

Our experience with using Serval provides opportunities for improving verification tools. While effective at identifying performance bottlenecks during symbolic evaluation, symbolic profiling requires manual efforts to analyze profiler output and develop symbolic optimizations (§6.4), and does not profile the SMT solver; automating these steps would reduce the verification burden for system developers. Another promising direction is to explore how to combine the strengths of different tools to verify a broader range of properties and systems [26, 68, 86].

9 Conclusion

Serval is a framework that enables scalable verification for systems code via symbolic evaluation. It accomplishes this by lifting interpreters written by developers into automated verifiers, and by introducing a systematic approach to identify and overcome bottlenecks through symbolic profiling and optimizations. We demonstrate the effectiveness of this approach by retrofitting previous verified systems to use Serval for automated verification, and by using Serval to find previously unknown bugs in unverified systems. We compare and discuss the trade-offs of various methodologies for verifying systems software, and hope that these discussions will be helpful for others making decisions on verifying their systems. All of Serval's source is publicly available at: https://unsat.cs.washington.edu/projects/serval/.

Acknowledgments

We thank Jon Howell, Frans Kaashoek, the anonymous reviewers, and our shepherd, Emmett Witchel, for their feedback. We also thank Andrew Waterman for answering our questions about RISC-V, David Kohlbrenner, Dayeol Lee, and Shweta Shinde for discussions on Keystone, and Daniel Borkmann, Palmer Dabbelt, Song Liu, Alexei Starovoitov, Björn Töpel, and Jiong Wang for reviewing our patches to the Linux kernel. This work was supported by NSF awards CCF-1651225, CCF-1836724, and CNS-1844807, and by VMware.

References

[1] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. 2010. Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE).* Edinburgh, United Kingdom, 71–85.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* Atlanta, GA, 175–188.

[3] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th ACM*

Symposium on Operating Systems Principles (SOSP). Monterey, CA, 311–327.

[4] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*. Cascais, Portugal, Article 71, 31 pages.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Survey* 51, 3, Article 50 (July 2018), 39 pages.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. Amsterdam, The Netherlands, 364–387.

[7] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, 335–348.

[8] William R. Bevier. 1989. Kit: A Study in Operating System Verification. *IEEE Transactions on Software Engineering* 15, 11 (Nov. 1989), 1382–1396.

[9] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. 2006. Putting it all together – Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer* 8, 4–5 (Aug. 2006), 411–430.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Amsterdam, The Netherlands, 193–207.

[11] Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (Oct. 2009), 263–288.

[12] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada, 917–934.

[13] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. In *Proceedings of the 2018 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Boston, MA, Article 149, 26 pages.

[14] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. 1995. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications* 29, 2 (Jan. 1995), 27–62.

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, 991–1008.

[16] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 906–909.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 209–224.

[18] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. Alexandria, VA, 322–335.

[19] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90.

[20] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 270–281.

[21] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 270–286.

[22] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, 18–37.

[23] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 265–278.

[24] Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, 609–622.

[25] Maria Christakis and Patrice Godefroid. 2015. Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing. In *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Mumbai, India, 373–392.

[26] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous formal verification of Amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*. Oxford, United Kingdom, 430–446.

[27] Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *TSE* 2, 3 (5 1976), 215–222.

[28] Jonathan Corbet. 2015. Post-init read-only memory. https://lwn.net/Articles/666550/.

[29] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, 857–874.

[30] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, 648–664.

[31] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, 337–340.

[32] Leonardo de Moura and Nikolaj Bjørner. 2010. Bugs, Moles and Skeletons: Symbolic Reasoning for Software Development. In *Proceedings of the 5th International Joint Conference on Automated Reasoning*. Edinburgh, United Kingdom, 400–411.

[33] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. 1998. *Extended Static Checking*. Research Report SRC-RR-159. Compaq Systems Research Center.

[34] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, WA, 101–114.

[35] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71.

[36] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 287–305.

[37] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[38] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (March 2012), 40–44.

[39] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*. Oakland, CA, 11–20.

[40] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, 595–608.

[41] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, 653–669.

[42] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, 1–17.

[43] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 165–181.

[44] Daniel Jackson and Jeannette Wing. 1996. Lightweight Formal Methods. *IEEE Computer* 29, 4 (April 1996), 20–22.

[45] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[46] Dejan Jovanović and Leonardo de Moura. 2012. Solving Non-linear Arithmetic. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*. Manchester, United Kingdom, 339–354.

[47] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[48] Gerwin Klein. 2009. Operating system verification—An overview. *Sādhanā* 34, 1 (Feb. 2009), 27–69.

[49] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–70.

[50] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, 207–220.

[51] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, 19–37.

[52] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 193–204.

[53] Leslie Lamport. 2008. Computation and State Machines.

[54] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. Palo Alto, CA, 75–86.

[55] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. 2019. Keystone: A Framework for Architecting TEEs. https://arxiv.org/abs/1907.10119.

[56] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Dakar, Senegal, 348–370.

[57] K. Rustan M. Leino and Michał Moskal. 2010. Usable Auto-Active Verification. In *Workshop on Usable Verification*. Redmond, WA, 4.

[58] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

[59] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA.

[60] Peng Li and Steve Zdancewic. 2005. Downgrading Policies and Relaxed Noninterference. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*. Long Beach, CA, 158–170.

[61] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, 973–990.

[62] Haohui Mai, Edgar Pek, Hui Xue, Samuel T. King, and P. Madhusudan. 2013. Verifying Security Invariants in ExpressOS. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, 293–304.

[63] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the 5th Workshop on Hardware and Architectural Support for Security and Privacy*. Seoul, South Korea, 9.

[64] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA, 415–429.

[65] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 252–269.

[66] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2016. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag.

[67] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Nara, Japan, 89–102.

[68] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*. Toronto, Canada, 23–41.

[69] Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the ARM v8-M Architecture Specification. In *Proceedings of the 2017 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vancouver, Canada, Article 88, 24 pages.

[70] John Rushby. 1992. *Noninterference, Transitivity, and Channel-Control Security Policies*. Technical Report CSL-92-02. SRI International.

[71] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, WA, 335–350.

[72] Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, 471–482.

[73] SiFive. 2019. *SiFive U54 Core Complex Manual, v19.05*. SiFive, Inc. https://www.sifive.com/cores/u54

[74] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, 1–16.

[75] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, 287–306.

[76] Venkatesh Srinivasan and Thomas Reps. 2015. Partial Evaluation of Machine Code. In *Proceedings of the 2015 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, 860–879.

[77] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow As a Safety Problem. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*. London, United Kingdom, 352–367.

[78] The Clang Team. 2019. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[79] The Coq Development Team. 2019. *The Coq Proof Assistant, version 8.9.0*. https://doi.org/10.5281/zenodo.2554024

[80] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Onward!* Boston, MA, 135–152.

[81] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541.

[82] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. -Overify: Optimizing Programs for Fast Verification. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, NM, 6.

[83] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 33–47.

[84] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, 260–275.

[85] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation.

[86] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Article 25, 28 pages.

[87] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Canada, 99–110.

[88] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 203–216.