Retrofitting automated verification to systems code by scaling symbolic evaluation

LUKE NELSON

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Washington

2025

Reading Committee:
Xi Wang, Chair
Emina Torlak
Jon Howell

Program Authorized to Offer Degree:
Computer Science & Engineering

UNIVERSITY OF WASHINGTON

Abstract

Retrofitting automated verification to systems code by scaling symbolic evaluation

Luke Nelson

Chair of the Supervisory Committee:
Xi Wang
Computer Science & Engineering

Formal verification is a technique for eliminating classes of bugs in systems software by formally proving that a system's implementation meets its intended specification. While effective at systematically preventing hard-to-catch bugs, formal verification demands a significant effort from developers in the form of manual proofs. Automated verification techniques reduce the burden of verification by leveraging automated reasoning to avoid the need for manual proofs. But as a result, they sacrifice generality and require developers to build bespoke verification tools and to carefully design systems with automated verification in mind.

This dissertation explores how to make it easier to build and reuse automated verifiers, and how to retrofit systems to automated verification. To do so, we built Serval, a framework for writing automated verifiers for systems code. To use Serval, developers write an interpreter for a language; Serval then leverages the Rosette programming language to lift the interpreter into a verifier via symbolic evaluation. Serval also comes with a set of techniques and optimizations to help overcome verification bottlenecks.

We use Serval to develop automated verifiers for RISC-V, x86, Arm, LLVM IR, and BPF. We apply these verifiers to retrofit automated verification to two existing security monitors previously formally verified using other techniques: CertiKOS [46], an OS kernel with strict process isolation, and Komodo [58], a monitor that implements secure enclaves. We port these two systems to RISC-V, modifying their interfaces for automated verification and to improve security. We write specifications amenable to automation, and compare our efforts with that of the original systems.

To demonstrate applicability to systems beyond security monitors, we use Serval to build Jitterbug, a framework for writing and verifying just-in-time (JIT) compilers for the Berkeley Packet Filter (BPF) language in the Linux kernel. We develop a specification of compiler correctness suitable for these JITs. Using this approach, we found and fixed more than 30 new bugs in the JITs in the Linux kernel and developed a new, verified BPF JIT for 32-bit RISC-V.

Acknowledgements

First, I want to give my utmost thanks to my advisor, Xi Wang. I first met Xi when I took a systems programming course he taught in 2015. I enjoyed that course so much that I signed up for his operating systems course the following fall, during which he invited me to work with him and one of his PhD students on a project that incorporated both programming languages and as low-level systems hacking; two things I was most interested in. We started to work together, and I enjoyed getting exposed to systems research through him. By the time I graduated, I knew I wanted to continue on at UW for grad school because of the opportunity to have him as my advisor. I have learned a lot from him about paper writing, system design, and life in general and I'm fortunate that I still get to work with and learn from him.

I also want to thank the other members of my committee, Emina Torlak, Jon Howell, and Payman Arabshahi. I am grateful to Emina for her formal methods expertise, and, of course, I couldn't have asked for a better person to help me debug my Rosette code. I've greatly enjoyed my many discussions with Jon about what the future of formal verification will look like (automated or otherwise).

The work in this dissertation would not have been possible without my labmates, coauthors, and friends James Bornholt, Helgi Sigurbjarnarson, and Jacob Van Geffen. I also want to thank my coauthors, Ronghui Gu and Andrew Baumann, as well as the anonymous reviewers of SOSP 2019 and OSDI 2020.

I have gotten to work with and learn from so many amazing students and faculty members at UW. In addition to everyone named previously, I want to thank my friends Jared Roesch, Zachary Tatlock, Sorawee Porncharoenwase, and Kaj Bostrom. It is a daunting task to come up with an exhaustive list of all the people I'm fortunate enough to have met at UW during the decade I spent there, and I won't presume to be able to do so; I am grateful for each and every one of them all the same. I also want to thank the graduate advising team, and Elise Dorough in particular, for helping me throughout my graduate career and moving mountains to get me to this point.

Lastly, and most importantly, I want to thank my family: my parents Eric and Michelle, and my sister Brooklyn. You have always supported me, and I've always been able to turn to you for advice, or just to have someone to talk to. I am incredibly proud to be your son and brother, and I'm grateful for everything you have done for me.

Portions of this thesis appeared in the following publications:

[1] Luke Nelson et al. “Scaling symbolic evaluation for automated verification of systems code with Serval”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, Oct. 2019, pp. 225–242.

[2] Luke Nelson et al. “Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel”. In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual conference, Nov. 2020, pp. 41–61.

Contents

Chapter 1

Introduction

Formal verification provides a general approach to proving critical properties of systems software [81]. 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 [82].

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 [82, §7.2].

The *push-button verification* approach [121, 155, 154] 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 [112] and discharges the query with a solver such as Z3 [114].

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 dissertation aims to answer these questions.

In chapter 2, we present Serval, an extensible framework for writing automated verifiers. In prior work on push-button verification [155, 121, 154], 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 [168, 169], an extension of the Racket language [56] 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 [160].

Using Serval, we build automated verifiers for RISC-V [179], x86-32, LLVM [91], and Berkeley Packet Filter (BPF) [59]. These verifiers are simple and easy to understand, and inherit from Rosette vital optimizations for free, such as constraint caching [29], partial evaluation [74], and state merging [88]. 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.

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 [8].

To address this challenge, Serval adopts recent advances in *symbolic profiling* [22], 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.

Automated verifiers restrict the types of properties and systems that can be verified in exchange for proof automation. In chapter 3, we explore what changes are needed in order to retrofit an existing system to automated verification and the limitations of this methodology are. To do so, we conduct case studies using two state-of-the-art verified systems: CertiKOS [46], a security monitor that provides strict isolation on x86 and is verified using the Coq interactive theorem prover [166], and Komodo [58], a security monitor that provides software enclaves on ARM and is verified using the Dafny auto-active theorem prover [94]. 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 [153] on a HiFive Unleashed development board. Like the original efforts, we prove the monitors' functional correctness through refinement [89] and higher-level properties such as noninterference [64]; 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 subsection 2.1.5).

In addition, Serval generalizes to use cases beyond standard refinement-based verification. We started by using Serval to write lightweight checkers for the BPF JIT compilers targeting RISC-V and x86-32 in the Linux kernel. Using these checkers, we find a total of 15 new bugs in these JIT compilers. In chapter 4, we describe how we extended these checkers to build Jitterbug, a framework for building and verifying BPF JIT compilers. Jitterbug includes a domain-specific language for writing JIT compilers, a specification of JIT correctness amenable to automated reasoning, and techniques for scaling verification to JIT compilers. We use Jitterbug to build a new BPF JIT for the 32-bit RISC-V architecture, and to find an additional 16 bugs in the existing JIT compilers. The new BPF JIT and the bug fixes for the existing BPF JITs have all been upstreamed to the Linux kernel.

Through this dissertation, 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 [45, 183, 151, 93], 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.

1.1 Related work

Interactive verification. There is a long and rich history of using interactive theorem provers to verify the correctness of systems software [81]. 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 [14], 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 [24].

The seL4 project [84, 82] 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 [125]. 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 [152], co-generating code and proof from a high-level language [2, 127], and a proof of noninterference [117].

CertiKOS presents a layered approach for verifying the correctness of an OS kernel with a mix of C and assembly code [67]. CertiKOS adapts CompCert [97], 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 [166]. An extension of CertiKOS achieves the first functional correctness proof of a concurrent kernel [66], which is beyond the scope of this paper. We use the publicly available uniprocessor version of CertiKOS, described by Costanzo, Shao, and Gu [46], 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 [36, 35]. 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 [15]; and Bedrock [38], which verifies web applications for robots down to the assembly level.

Auto-active verification. In contrast to interactive theorem provers, auto-active theorem provers [95] ask developers to write proof annotations [48] 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 [103] and type safety in the Verve OS [180]. Ironclad [68] 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 [94]; next, an untrusted compiler translates the implementation to a verifiable form of x86 assembly, which the Boogie verifier [9] 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 [58] 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 [20], 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 [155], Hyperkernel [121], and extensions to information-flow control using Nickel [154].

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 [169, 168], which powers a range of symbolic reasoning tools [22, §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 [41, 80] and bounded model checking [16] can be used to find bugs in systems code, using tools like KLEE [27], S2E [37], and SAGE [63]; see Baldoni et al. [8] 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 [39] and verifying software dataplanes using S2E [49]. 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 [88] and transformations of implementation code [171, 26]. Serval uses symbolic profiling [22] to identify performance bottlenecks in verifiers and provides symbolic optimizations that exploit domain knowledge to repair these bottlenecks.

Chapter 2

Building automated verifiers with Serval

This chapter describes Serval, a framework for developing automated verifiers for systems software[1]. 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.

2.1 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 (subsection 2.1.1), followed by an example of how to write, profile, and optimize a verifier for a toy instruction set (subsection 2.1.2), and how to prove properties (subsection 2.1.3). Next, we describe Serval's support for verifying systems code (subsection 2.1.4). We end this section with a discussion of limitations (subsection 2.1.5).

2.1.1 Overview

Figure 2.1 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.

[1]Portions of this chapter and the following chapter were first published in the proceedings of SOSP 2019 as *Scaling symbolic evaluation for automated verification of systems code with Serval*, by Nelson, Bornholt, Gu, Baumann, Torlak, and Wang [122].

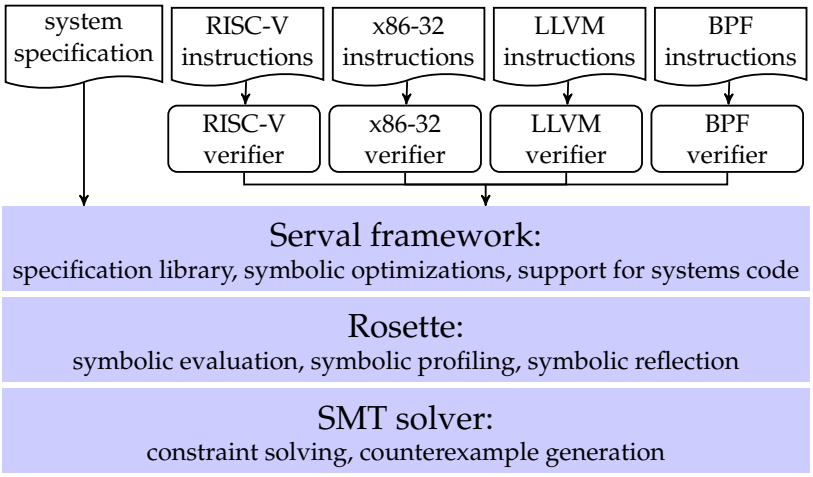

FIGURE 2.1: The Serval verification stack. Curved boxes denote verification input and rounded-corner boxes denote verifiers.

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 [121], since otherwise symbolic evaluation diverges. When symbolic evaluation is slow or hangs, system developers invoke the Rosette symbolic profiler [22]. 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 2.1 to scale to all the systems studied in this paper. Symbolic optimizations examine the structure of symbolic values (via *symbolic reflection* [168, §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 (section 3.5). 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.

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 2.2: 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 2.3: A ToyRISC program for computing the sign of a_0.

2.1.2 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 2.2). 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 2.3 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.

Writing an interpreter. Figure 2.4 lists the full ToyRISC interpreter. It is written in the Rosette language. The syntax is based on S-expressions. For example, `(+ 1 pc)` evaluates to the value of one plus that of `pc`. The expression `(define (name args) body ...)` defines a function with the given name and list of arguments and evaluates to the value 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.

```
#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 2.4: A ToyRISC interpreter using Serval (in Rosette).

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 on the instructions from Figure 2.3 with initial state $pc = 0, a_0 = 42, a_1 = 0$ results in $pc = 0, a_0 = 1, a_1 = 0$.

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
```

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 2.5 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 [28], Rosette relies on two basic strategies: symbolic execution [41, 80] and bounded model checking [16]. 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 [88]. Rosette employs a hybrid strategy [168], which works well in most cases. For instance, after executing `sltz` in Figure 2.5, 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 [22]. 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 2.4 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.

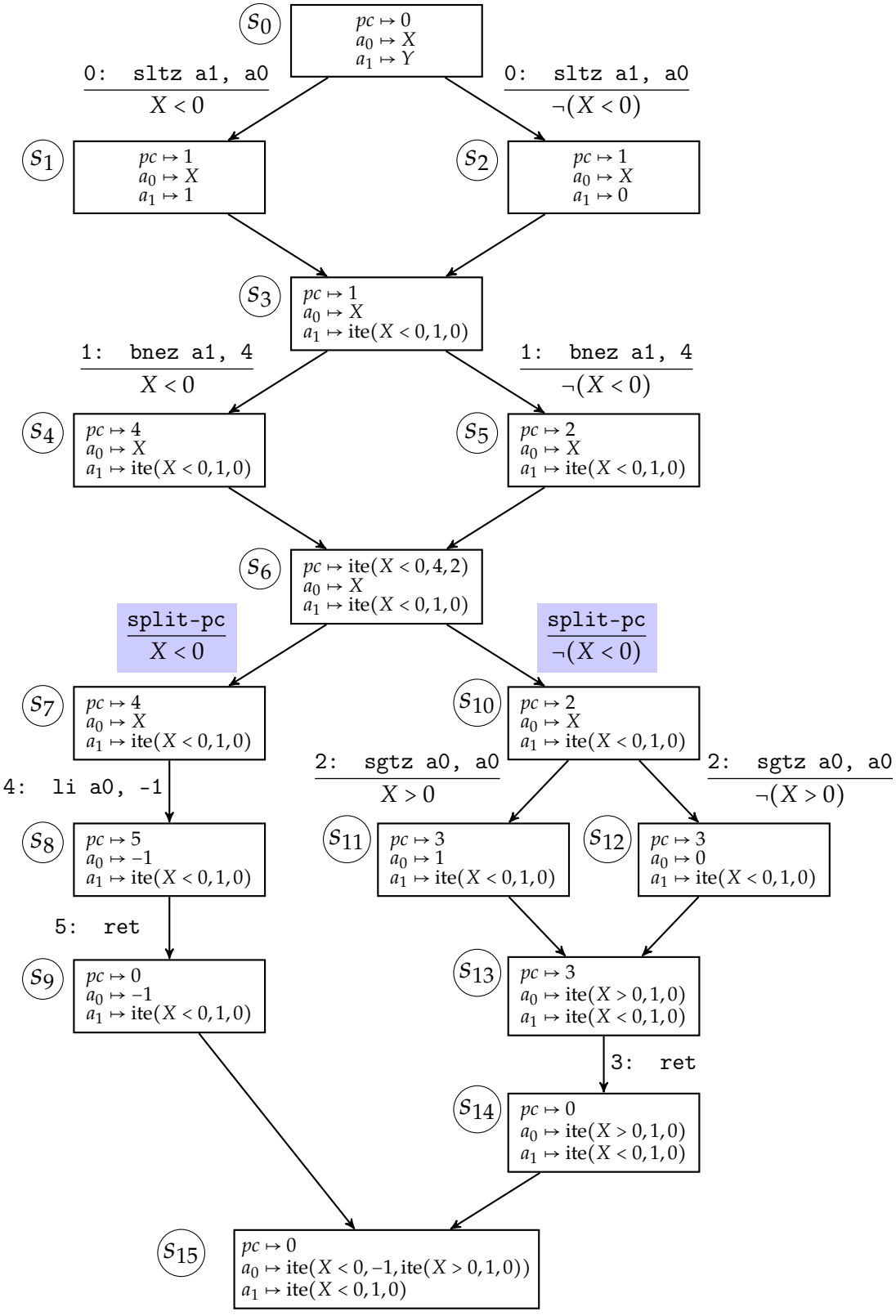

FIGURE 2.5: Symbolic evaluation of the *sign* program (Figure 2.3) using the ToyRISC interpreter (Figure 2.4).

Diagnosing performance bottlenecks. Suppose that the code in Figure 2.4 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 [22] 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 [75]; one may adopt practice from prior verification efforts to sidestep such issues [121, 58, 68].

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 (section 2.2).

2.1.3 Verifying properties

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

Absence of undefined behavior. As shown in Figure 2.4, 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 [176, §3.2.1]. Serval's LLVM verifier also reuses checks inserted by Clang's UndefinedBehaviorSanitizer [165] 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 [89]. 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 2.4) 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 2.3, 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 [147]. Safety properties are predicates on specification states. This chapter 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 [163]. Serval provides definitions of common one- and two-safety properties, such as reference-count consistency [121, §3.3] and noninterference properties [64], 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* [149], which asks for an *unwinding relation* $\sim$ 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: $s_1 \sim s_2 \Rightarrow$ `spec-sign`$(s_1) \sim$ `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`).

2.1.4 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 2.6 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 [82, 58, 46, 121, 154], we assume that the system runs on a single core, with interrupts disabled in privileged mode; therefore, each trap handler runs in its entirety.

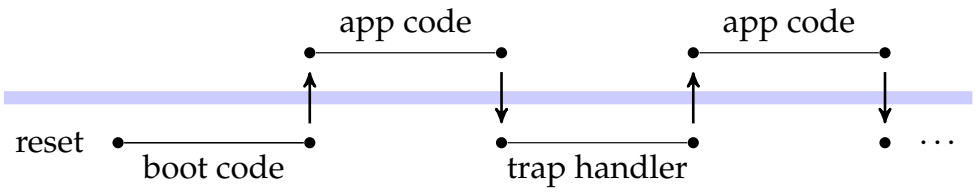

FIGURE 2.6: System execution: the lower half and the higher half denote execution in privileged and unprivileged modes, respectively; and arrows denote privilege-level transitions.

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 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 [175, §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 [27] and CompCert [98]. 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`.

2.1.5 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 (section 3.4).

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 [4] 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 chapter 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 [101, 85, 25]. 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 chapter satisfy this restriction. In addition, it cannot specify properties outside the decidable fragment of first-order logic supported by the specification library (subsection 2.1.1). 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 chapter 3 we describe such examples and alternative specifications that are expressible in Serval.

2.2 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 (section 3.4).

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 2.5 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 [88], for example, by selectively applying `split-pc` to certain program fragments.

Symbolic memory addresses. When a memory address is symbolic (e.g., due to a computed memory offset or integer-to-pointer conversion), 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 [27]; some prior verifiers simply disallow integer-to-pointer conversions [121, §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: $(C_0 \times \mathit{pid} + C_1) \bmod C_0$. 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* [22]: 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 $(C_0 \times \mathit{pid} + C_1) \bmod C_0$, 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 (subsection 2.1.4); 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 (subsection 2.1.4), 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 [44], 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.

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 2.7: Lines counts of the Serval framework and verifiers.

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 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:

$$\text{ite}(x = C_0, C_0, \text{ite}(x = C_1, C_1, \ldots \text{ite}(x = C_{n-1}, C_{n-1}, x) \ldots))$$

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.

2.3 Implementation

Figure 2.7 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 [121, §3.2]. The BPF verifier implements the extended BPF instruction set [59], with limited support for in-kernel helper functions.

As a comparison, prior push-button LLVM verifiers [121, 154] 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.

Chapter 3

Retrofitting systems for automated verification

Chapter 2 described how to use Serval to build automated verifiers. This chapter describes how to apply those verifiers to retrofit automated verification to existing systems. As case studies, 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. We report our changes and discusses the trade-offs of the three verification methodologies.

3.1 Protection mechanisms on RISC-V

As shown in Figure 3.1, both CertiKOS^s and Komodo^s run in machine mode (M-mode), the most privileged mode on 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 [11, 121].

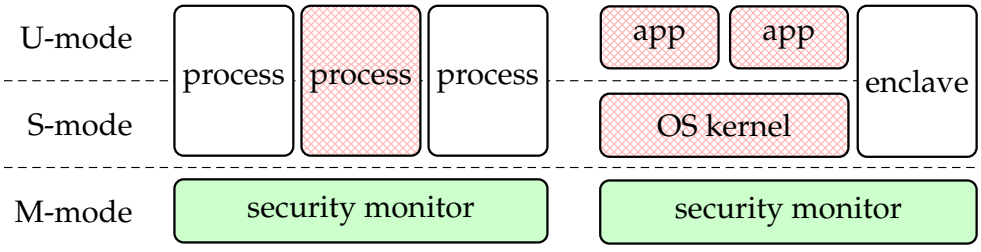

FIGURE 3.1: Two security monitors ported to RISC-V (shaded boxes), which aim to protect legitimate components (white boxes) from adversaries (crosshatched boxes).

The monitors utilize the *physical memory protection* (PMP) and *trap virtual memory* (TVM) protection mechanisms of RISC-V [179]. 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 on memory accesses from S- or U-mode. TVM allows M-mode to trap accesses made by 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 that runs 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 that runs in S- or U-mode.

We do not consider direct-memory-access (DMA) attacks, 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 [151] to be sufficient once an IOMMU is available.

3.2 CertiKOS

CertiKOS as described by Costanzo, Shao, and Gu [46] 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.

Security. CertiKOS formalizes its security using noninterference, specifically, step consistency (subsection 2.1.3). 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 3.2 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).

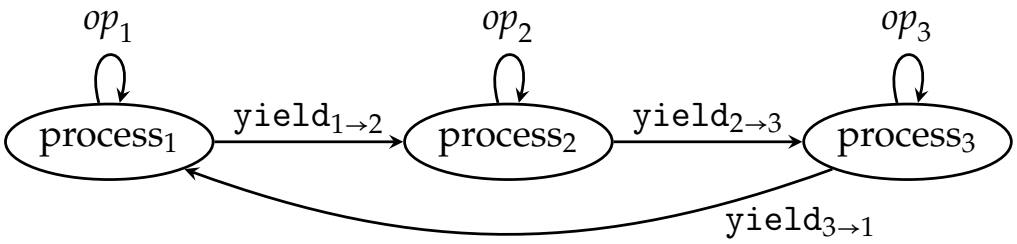

FIGURE 3.2: Actions by processes in CertiKOS; op_i denotes either `get_quota`, `spawn`, or a memory access by process i.

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 $s_1 \overset{p}{\sim} s_2$, to mean that the portions of the states observable to p are the same. Let $\text{step}(s, a)$ 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: $s_1 \overset{p}{\sim} s_2 \Rightarrow \text{step}(s_1, a) \overset{p}{\sim} \text{step}(s_2, a)$.

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 [97] to translate it into a Coq representation of an abstract syntax tree (AST) in the Clight intermediate language [19]. 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 [46, §5]. The three properties are the following, using process_1 in Figure 3.2 as an example:

- If process_1 performs a small-step action (op_1 or $\texttt{yield}_{1\to2}$) from two indistinguishable states, the resulting states must be indistinguishable.
- If any process other than process_1 performs a small-step action (op_2, $\texttt{yield}_{2\to3}$, or op_3), the states before and after the action must be indistinguishable to process_1.
- If process_1 is yielded to ($\texttt{yield}_{3\to1}$) 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. [67] for details. Below we describe two examples. One, CompCert models memory as an infinite number of blocks [98]. 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 [31].

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 [121, §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 computes the child PID as $N \times pid + nr\_children + 1$, 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 discloses 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 subsection 2.1.4. 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 (subsection 2.1.5). 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 [154]; 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 [149, 117], which enabled us to catch the PID covert channel in `spawn`.

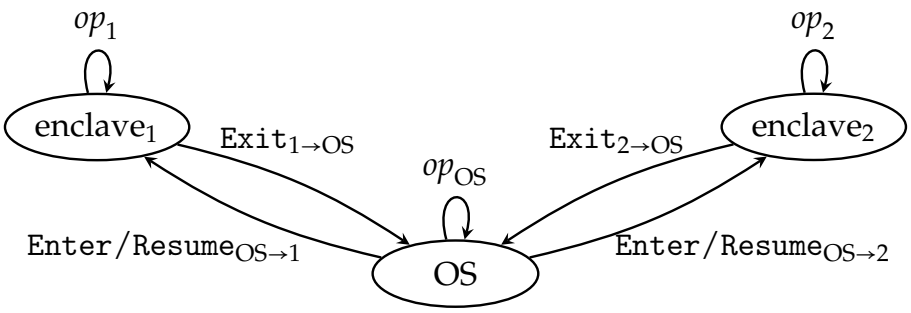

FIGURE 3.3: Actions by enclaves and the OS in Komodo; op_i and op_{OS} denote either monitor calls or memory accesses by enclave i and by the OS, respectively.

3.3 Komodo

Komodo [58] 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 (section 3.2), 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 "$\texttt{Enter}_{OS\to 1}; op_1^*; \texttt{Exit}_{1\to OS}$" in Figure 3.3 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: $s_1 \overset{L}{\sim} s_2 \Rightarrow \text{step}(s_1, a) \overset{L}{\sim} \text{step}(s_2, a)$.

A subtlety is that Komodo permits legitimate information flows (e.g., intentional declassification [100]) 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 [94] and Vale [20] 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.

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 [109], 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 [82], Hyperkernel [121], 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 (section 3.1).

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 (subsection 2.1.3). 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 as with 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 [154]. 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.

	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 3.4: Sizes and verification times of the monitors.

3.4 Results

Figure 3.4 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.

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 [121, 154]. 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 [121, §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 subsection 2.1.2; 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 [152]. 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 subsection 2.1.5, 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 [6], 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.

3.5 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 [58, §9] and repair (e.g., requiring use of triggers [69, §6]). Building on Rosette, Serval chooses to limit system specifications to a decidable fragment of first-order logic (subsection 2.1.1) 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 [22].

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 section 3.2 and section 3.3, 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.

3.6 Finding bugs via verification

Besides proving refinement and noninterference properties, we also apply Serval to write and prove *partial* specifications [73] 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 [93], 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 (section 3.3). 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 [59]. 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 [175].

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. We later expanded on the checker work to build automated verifiers for BPF JIT compilers, which we describe in chapter 4.

3.7 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 [168, §2.3], is a good match for implementing symbolic optimizations.

Our experience with using Serval has identified opportunities for improving verification tools. For example, when we built Serval, symbolic profiling still required manual effort to analyze profile output and to develop symbolic optimizations, which can be difficult in general. Since then, SymFix [131] demonstrated the feasibility of finding symbolic optimizations automatically via search, using Serval as one case study. Tooling improvements like this reduce the developer burden of using automated verification tools even further.

Chapter 4

Scaling automated verifiers to BPF JIT compilers

This chapter describes our experience applying Serval to a a critical component in the Linux kernel, the just-in-time compilers ("JITs") for the Berkeley Packet Filter (BPF) virtual machine[1]. We verify these JITs using Jitterbug, the first framework to provide a precise specification of JIT correctness that is capable of ruling out real-world bugs, and an automated proof strategy that scales to practical implementations. Using Jitterbug, we have designed, implemented, and verified a new BPF JIT for 32-bit RISC-V, found and fixed 16 previously unknown bugs in five other deployed JITs, and developed new JIT optimizations; all of these changes have been upstreamed to the Linux kernel. The results show that it is possible to build a verified component within a large, unverified system with careful design of specification and proof strategy.

4.1 Introduction

Downloading application code into the OS kernel is a general approach to extensibility [53]. To extend the kernel, the application submits a program written in a dedicated language, and the kernel executes this program using an interpreter, or translates it into machine code for native execution via a *just-in-time (JIT) compiler* [7]. Berkeley Packet Filter (BPF) [59] is one such language, and it is used to implement a wide variety of extensions for the Linux kernel, including networking [71], security [156], and tracing [65], among many other services [104, 42].

Given the prevalence of BPF code and its execution in the OS kernel, the correctness of BPF JIT compilers (or simply "JITs") is critical for the system. Compared to the BPF interpreter, using the JITs is both more efficient and more resistant to speculative attacks [161], leading major Linux distributions to remove the BPF interpreter from the

[1]Portions of this chapter were first published in the proceedings of OSDI 2020 as *Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel*, by Nelson, Geffen, Torlak, and Wang [123].

kernel in favor of the JITs [21]. But the JITs are more susceptible to subtle correctness bugs due to their complexity (section 4.3).

This chapter presents a formal approach to building JITs in the kernel with high assurance of correctness. We develop Jitterbug, a framework for writing JITs and proving them correct. Using Jitterbug, we design, implement, and verify a BPF JIT for RV32, the 32-bit RISC-V architecture [178]. We also port the existing JITs for Arm32, Arm64, RV64, x86-32, and x86-64 to Jitterbug, uncovering 16 previously unknown bugs. We write patches that fix these bugs and introduce new optimizations, all of which are verified to be correct. The BPF JIT for RV32, bug fixes, and optimizations have been upstreamed to the Linux kernel.

Jitterbug is designed to meet three competing requirements: *deployability* of verified JITs with minimal changes to the Linux kernel; *proof automation* to support rapid verification of JITs; and *separability* of verified JITs from any verification artifacts, making the resulting code auditable by kernel developers with no background in formal methods. Each of these requirements comes with its own challenges and trade-offs.

First, BPF JITs and their generated code interact with a monolithic kernel via an existing interface, which was not designed for verification. As Jitterbug emphasizes deployability, it cannot adopt the clean-slate design favored by previous verification efforts [61, 118, 158, 175] or change this interface to simplify verification. Therefore, it needs a correctness specification that is both capable of ruling out real-world bugs and amenable to verification. Developing such a specification is challenging even for clean-slate designs with strong simplifying assumptions, and it is the core technical challenge addressed by Jitterbug.

Second, verification needs to catch up with increasing functionality and optimization of BPF JITs. Jitterbug thus prioritizes proof automation to free developers from the burden of writing manual proofs and to enable rapid verification in the code review process. Prior work has shown success in scaling automated verification to systems whose code does not change in response to input [122, 126]. But verifying a JIT is particularly challenging, because it requires reasoning about not only the behavior of the JIT itself, but also that of the machine code generated by the JIT for input BPF programs.

Third, kernel development emphasizes the efficiency and clarity of source code, whereas formal development emphasizes managing code complexity to make verification tractable. Jitterbug must resolve the tension and make the two development processes cleanly separable. While formal development can use specific tools and artifacts such as specifications, the final implementation of a JIT needs to be C code that can be reviewed assuming no knowledge of formal methods, and can be compiled using a standard toolchain.

To address these challenges, Jitterbug makes the following contributions:

- A precise stepwise specification for JIT correctness (section 4.4). The specification models both BPF and target architectures as abstract machines, and it formulates JIT correctness as the behavioral equivalence of running the machines with a source BPF instruction and the target instructions produced by the JIT, respectively. The specification assumes that a JIT translates a single source instruction at a time. This assumption matches real-world BPF JIT implementations and obviates the need to reason about translating entire programs.
- An automated proof strategy that scales to practical BPF JITs (section 4.5). Building on Serval [122], Jitterbug uses symbolic evaluation [168, 22] to produce a satisfiability query that encodes the semantics of a JIT implementation, the semantics of source BPF code, and the semantics of target machine code produced by the JIT. It then discharges the query using an SMT solver [113]. Since Serval was designed to reason about systems whose code is statically known, it cannot be used to verify *symbolic* instructions (e.g., with symbolic fields, at symbolic addresses) generated by the symbolic evaluation of a JIT. Jitterbug addresses this challenge with a symbolic evaluation strategy that can reason about such symbolic code.
- An approach to writing JITs in a domain-specific language (DSL) based on C (section 4.6). The Jitterbug DSL is a *shallow embedding* of a structured subset of C in the Rosette programming language [168, 169], which extends Racket [57] for symbolic reasoning. That is, the Jitterbug DSL implements a subset of C as a Rosette library. We write new JITs in the DSL, which simplifies verification and enables synthesis of JIT optimizations [159, 106]. Jitterbug automates the step of translating JITs written in the DSL to C through an (unverified) extraction mechanism. We verify existing JITs by manually translating their C code to Rosette.
- Experience with using Jitterbug to build a BPF JIT for RV32, find and fix bugs in five existing BPF JITs, perform code review, develop optimizations, and port a JIT for a stack machine [118], all with low verification overhead (section 4.7). One of the bugs has led to a clarification in the RISC-V instruction-set manual. We report on the iterative process of improving Jitterbug and upstreaming JIT code to the Linux kernel.

To our knowledge, Jitterbug is the first to provide a specification that rules out bugs in practical JIT implementations, and a proof strategy that scales automated verification to a class of compilers. It demonstrates the feasibility of building a verified component (i.e., the BPF JIT) within a large, unverified system under active development (i.e., the Linux kernel), through careful design of specification and proof strategy.

4.2 Related work

Code downloading for extensible systems. The Xerox Alto allows applications to customize and optimize the system through *microcode* [164, 90]. It pioneered the use of

packet filters for demultiplexing, debugging, and monitoring.

The CMU/Stanford Packet Filter [110] introduced a *stack-based* virtual machine into the 4.3BSD kernel to interpret packet filters. To enable more efficient implementations, the Berkeley Packet Filter (BPF) [108] adopts a *register-based* virtual machine instead, which consists of two 32-bit registers and a scratch memory. BPF has gained a wide adoption in BSD and Linux kernels. Besides BPF, DTrace [30] and Lua on NetBSD [170] are two other in-kernel virtual machines.

A redesign of BPF in the Linux kernel started in 2014, first as an optimization of the internal representation of BPF instructions for 64-bit architectures [162]. It has since grown into a full RISC-like virtual machine, with 64-bit general-purpose registers, flexible control flow (e.g., bounded loops and BPF-to-BPF calls), and safe access to kernel memory. The generality and expressiveness have led to an explosion of tools and systems based on BPF, ranging from networking [71], security [156], tracing [65], to storage [17], virtualization [3, 128], and hardware offloading [79]. The new design is also called "extended BPF" or simply "BPF" in the Linux kernel, while the original design is referred to as *classic* BPF to avoid ambiguity. Unless otherwise noted, we follow this terminology and use BPF to refer to the new design. This chapter focuses on building verified JITs for BPF.

More generally, the exokernels [53] demonstrate a diverse set of mechanisms for code downloading, such as accelerating packet filtering using JIT compilation [52], sandboxing machine code [173] using software-based fault isolation [172, 150], and analyzing file-system metadata using an in-kernel virtual machine [76]. Other extensibility mechanisms include using safe languages [13, 55, 99] and proof-carrying code [120].

Correctness of JIT compilation. Just-in-time compilation (JIT) is a well-studied dynamic code generation technique dating back to Lisp [7, 78] and regular expressions in the QED text editor [148, 167]. It has also been used for dynamically typed languages [33], emulators [12], and specialization [130, 107].

This dissertation considers JITs that are realized as *static* compilers, using static register allocation and performing no garbage collection for memory management. In contrast to sophisticated dynamic code generation systems such as those for Java or JavaScript, this simplicity makes static JITs applicable to a restricted environment such as the kernel [51].

There is a rich literature on compiler correctness. Readers may refer to Young [182] and Leroy [96] for overviews. Compilers, especially optimizing compilers, can have multiple intermediate representations and translation passes, whereas the JITs considered in this dissertation are much simpler and resemble a one-pass compiler. On the other hand, compilers usually output assembly code, relying on a separate assembler and linker (e.g., GNU `as` and `ld`) to produce final machine code. The JITs run in the

kernel and directly produce machine code, effectively combining a compiler, assembler, and linker.

The closest efforts in this area are the verified JITs by Myreen [118] and Jitk [175]. The former translates code in a simple stack-based instruction set to x86-32 (see section 4.7), and is verified using the HOL4 theorem prover [157]. The JIT is implemented in HOL4 and translated to x86-32 machine code by a separate compiler [119]. Jitk builds on the CompCert verified compiler [97] to translate classic BPF to assembly, and is verified using the Coq theorem prover [166]. The JIT is implemented in Coq and extracted to OCaml code; it runs in user space rather than in the kernel due to the dependency on the OCaml runtime, an assembler, and a linker. Both efforts employ clean-slate designs, require manual proofs, and do not have a C implementation. Jitterbug is inspired by these efforts and shares the goal of building verified JITs, but prioritizes applicability to existing systems, proof automation, and implementation that can be reviewed independent of verification.

Compiler testing and fuzzing tools employ effective strategies to randomly generate input programs and check for miscompilation [105]. Csmith [181] and EMI fuzzers [92] have been used to find hundreds of bugs in GCC and LLVM. Kernel fuzzers such as syzkaller and trinity support generation of random BPF programs [50]. Serval [122] implements a bug finder for the compilation of BPF arithmetic and bitwise instructions. These tools generally do not exhaust all execution paths, thus providing no correctness guarantees for JITs.

Designing verified systems for deployment. Deployability is a desirable goal for formally verified systems, but it requires navigating an extra set of design trade-offs. As the first verified general-purpose microkernel, seL4 [82] pioneered many aspects of the design and deployment processes. For instance, it introduced a Haskell prototype as the bridge between formal methods and kernel developers, separating verification artifacts from the C implementation [84]. It has been deployed as a hypervisor to retrofit unverified, legacy software to power safety-critical systems [83, 70]. Another example is CompCert, the first verified C compiler. It has been integrated into the development process of control software for safety-critical systems [97, 77], replacing unverified compilers that were configured to disable optimizations due to risk concerns.

Cryptographic libraries are an attractive target for verification due to their essential role in security. For example, verified code from EverCrypt/HACL* [146, 184] and Fiat-Crypto [54] is used by Mozilla and Google, respectively. Amazon's s2n TLS implementation [40] is verified via a combination of manual and automated proofs.

Jitterbug presents a case study in applying formal methods to the BPF JITs in the Linux kernel. It shares these design challenges and addresses them with a precise specification and a proof strategy that scales to practical JIT implementations.

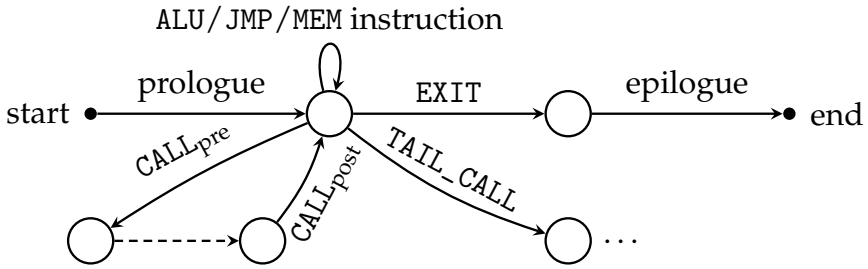

FIGURE 4.1: Transitions during the execution of a BPF program.

4.3 Case study

This section presents a brief overview of BPF and a case study of the BPF JIT bugs in the Linux kernel, which helped motivate the design of Jitterbug.

4.3.1 An overview of BPF

The BPF virtual machine consists of 12 explicit 64-bit registers: general-purpose registers `R0`–`R9`, a frame pointer `R10` that points to a stack memory region, and an internal register `AX` used by the kernel for rewrites (e.g., constant blinding against JIT spraying attacks [18]). It maintains a program counter `PC` and a tail-call counter `TCC`; the latter bounds the number of tail calls (to another BPF program without returning).

Currently, there are a total of 115 instruction opcodes, which can be categorized into the following:

- `ALU` (arithmetic and bitwise) instructions,
- `JMP` (unconditional and conditional jump) instructions,
- `MEM` (1-, 2-, 4-, and 8-byte memory access, and 4- and 8-byte atomic exchange-and-add) instructions,
- `CALL` to a kernel function or another BPF program; and
- `TAIL_CALL` and `EXIT`, which transfer control to another BPF program and the kernel, respectively.

Figure 4.1 depicts the execution of a BPF program. The input to a BPF program is provided by the kernel. Prologue and epilogue refer to initialization and cleanup code, respectively, for bridging the kernel. The BPF calling convention specifies that `R0` holds the return value, `R1`–`R5` pass arguments, and `R6`–`R9` are preserved across the call.

User processes may share data with BPF programs by creating BPF *maps* in the kernel, which are key/value stores of different data types. Maps may be accessed concurrently by BPF programs and user processes. Though there have been discussions, BPF has so far chosen not to specify a memory consistency model to avoid performance penalties [43].

Each BPF program consists of a sequence of instructions in *bytecode* (GCC/LLVM can compile C code to BPF). Upon receiving a BPF program from user space, the kernel invokes a checker to analyze whether the program is safe (e.g., free of division by zero, unbounded loops, and uninitialized register accesses) [62]; we refer to it as the BPF *checker* (rather than "BPF verifier" as by the Linux kernel to avoid ambiguity). If the BPF checker deems the program safe, the kernel invokes the JIT for compilation and attaches the resulting machine instructions to various hook points in the kernel for execution; otherwise, the kernel rejects the program. The JIT therefore considers safe programs only.

4.3.2 Bugs in BPF JITs

We manually inspected every commit to the BPF JITs in the Linux kernel from May 2014 (when the new BPF design was introduced) to April 2020, and categorized those that fixed JIT correctness bugs for Arm32, Arm64, RV64, x86-32, and x86-64; those for RV32 will be discussed in section 4.7. We consider "correctness bugs" as JITs producing erroneous machine instructions, and exclude non-correctness bugs (e.g., memory leaks during JIT compilation) from the study. In total, there are 41 commits that fixed 82 JIT correctness bugs during this period. See section A.2 for a complete list.

Below we describe some representative bugs we have found using Jitterbug. These bugs are difficult to find even for veteran developers, and were not caught by the existing test suite. They can lead to security vulnerabilities, since the resulting machine instructions run in the kernel and may process input from untrusted sources. For clarity, BPF instructions and registers are in uppercase, while target machine ones are in lowercase.

```
/* rd[0]: upper 32 bits of the destination register
 * rd[1]: lower 32 bits of the destination register
 * tmp2[1]: a temporary register */
if (val < 32) {
        /* tmp2[1] = rd[1] >> val */
        emit(ARM_MOV_SI(tmp2[1], rd[1], SRTYPE_LSR, val), ctx);
        /* rd[1] = tmp2[1] | (rd[0] << (32 - val)) */
        emit(ARM_ORR_SI(rd[1], tmp2[1], rd[0], SRTYPE_ASL,
                        32 - val), ctx);
        /* rd[0] = rd[0] >> val */
        emit(ARM_MOV_SI(rd[0], rd[0], SRTYPE_LSR, val), ctx);
} else if (val == 32) {
        /* rd[1] = rd[0] */
        emit(ARM_MOV_R(rd[1], rd[0]), ctx);
        /* rd[0] = 0 */
        emit(ARM_MOV_I(rd[0], 0), ctx);
} else {
        /* rd[1] = rd[0] >> (val - 32) */
        emit(ARM_MOV_SI(rd[1], rd[0], SRTYPE_LSR,
                        val - 32), ctx);
        /* rd[0] = 0 */
        emit(ARM_MOV_I(rd[0], 0), ctx);
}
```

FIGURE 4.2: Incorrect result with zero `val` for `RSH64_IMM` (Arm32).

Subtle architectural semantics. Figure 4.2 shows an excerpt of the Arm32 JIT for compiling `RSH64_IMM`, the BPF logical right shift instruction of a 64-bit register by an immediate. Since the target architecture is 32-bit, the JIT uses two machine registers, represented by `rd[0]` and `rd[1]`, to hold the upper and lower 32 bits of a 64-bit BPF register, respectively. The BPF checker ensures that the shift amount `val` is within the range $[0, 63]$. The emitted instructions work as follows:

- when the shift amount `val` is less than 32, the result of the upper half is simply `rd[0] >> val`, and the result of the lower half is `rd[1] >> val` combined with the bits shifted from the upper half, `rd[0] << (32 - val)`;
- the result of the upper half is simply zero, as all the bits are shifted out, and the result of the lower half holds the bits shifted from the upper half.

One subtlety in Arm32 is that a zero immediate in the `lsr` (logical shift right) instruction means right-shift by 32 bits (i.e., shifting all bits out) [5, §F5.1.103]. Therefore, when the shift amount `val` is zero, the instructions produced by the JIT incorrectly set the destination register to zero, instead of behaving as a no-op. This is further complicated by inconsistent semantics in Arm32: a zero immediate in the shift left instruction means a no-op. We fixed the bug by changing the JIT to emit no instructions when `val` is zero.

```
/* check if rvoff is in the range [-2^31,2^31 - 1] */
if (!is_32b_int(rvoff))
        return -ERANGE;
...
s64 upper = (rvoff + (1 << 11)) >> 12;
s64 lower = rvoff & 0xfff;
/* aupic t1,upper */
emit(rv_auipc(RV_REG_T1, upper), ctx);
/* jalr ra,lower(t1) */
emit(rv_jalr(RV_REG_RA, RV_REG_T1, lower), ctx);
```

FIGURE 4.3: Incorrect range check on `rvoff` for `CALL` (RV64).

Figure 4.3 shows another subtle bug in the RV64 JIT. Using a pair of `auipc+jalr` instructions is a standard way to support `pc`-relative call with a 32-bit offset on RISC-V [178]:

- `auipc t1,imm20` appends 12 low-order zero bits to a 20-bit immediate, sign-extends the 32-bit value to 64 bits, adds the sign-extended value to the address of the instruction, and writes the result in register `t1`;
- `jalr ra,imm12(t1)` jumps to a target address obtained by adding a sign-extended 12-bit immediate to the register `t1` and clearing the least-significant bit of the result for alignment; the address of the instruction following `jalr` is written to register `ra`.

One misconception about these instructions is that the `auipc+jalr` pair can reach any 32-bit offset in the range $[-2^{31}, 2^{31} - 1]$ on 64-bit RISC-V (RV64), by using certain `imm20` and `imm12` values. Part of the confusion stems from the "RV32I base integer instruction set" chapter in the RISC-V instruction-set manual indicating that `auipc+jalr` "can jump anywhere in a 32-bit pc-relative address range." But the same does not hold on RV64: both `auipc` and `jalr` *sign-extend* their results to 64 bits, causing the reachable offset range to shift by -2^{11}. Therefore, the range check on `rvoff` in the JIT is incorrect, which can lead to an off-target jump.

Our report prompted the RISC-V instruction-set manual to add the following clarification: "Note that the set of address offsets that can be formed by pairing LUI with LD, AUIPC with JALR, etc. in RV64I is $[-2^{31} - 2^{11}, 2^{31} - 2^{11} - 1]$." We fixed the bug in the JIT by using the clarified range for checking `rvoff`.

Subtle machine state. Figure 4.4 shows an excerpt of the x86-32 JIT for compiling BPF's `JSET64_REG` and `JSET32_REG` (in the form `BPF_JMP[32]|BPF_JSET|BPF_X` in C). The semantics of "`JSET64_REG DST,SRC,OFF`" is to perform a conditional jump when `DST&SRC` ("bitwise and" of two 64-bit BPF registers) is non-zero and fall through otherwise; the semantics of "`JSET32_REG DST,SRC,OFF`" is similar, using only the lower 32 bits of both `DST` and `SRC`.

Due to the limited number of registers on x86-32, the JIT spills some BPF registers on the stack. For simplicity, suppose that both `DST` and `SRC` are on the stack (i.e., both `dstk`

```
case BPF_JMP | BPF_JSET | BPF_X:
case BPF_JMP32 | BPF_JSET | BPF_X:
        bool is_jmp64 = BPF_CLASS(insn->code) == BPF_JMP;
        u8 dreg_lo = dstk ? IA32_EAX : dst_lo;
        u8 dreg_hi = dstk ? IA32_EDX : dst_hi;
        u8 sreg_lo = sstk ? IA32_ECX : src_lo;
        u8 sreg_hi = sstk ? IA32_EBX : src_hi;
        if (dstk) {
                EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EAX),
                      STACK_VAR(dst_lo));   /* eax <- dst_lo */
                if (is_jmp64)
                        EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EDX),
                              STACK_VAR(dst_hi)); /* edx <- dst_hi */
        }
        if (sstk) {
                EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_ECX),
                      STACK_VAR(src_lo));   /* ecx <- src_lo */
                if (is_jmp64)
                        EMIT3(0x8B, add_2reg(0x40, IA32_EBP, IA32_EBX),
                              STACK_VAR(src_hi)); /* ebx <- src_hi */
        }
        /* and dreg_lo,sreg_lo */
        EMIT2(0x23, add_2reg(0xC0, sreg_lo, dreg_lo));
        /* and dreg_hi,sreg_hi */
        EMIT2(0x23, add_2reg(0xC0, sreg_hi, dreg_hi));
        /*  or dreg_lo,dreg_hi */
        EMIT2(0x09, add_2reg(0xC0, dreg_lo, dreg_hi));
        goto emit_cond_jmp;   /* emit conditional jump */
```

FIGURE 4.4: Incorrect `eflags` value for `JSET32_REG` (x86-32).

and `sstk` are true). In this case, the JIT emits instructions to load the lower 32 bits of `DST` and `SRC` to `eax` and `ecx`, respectively. It also emits instructions to load the upper 32 bits to `edx` and `ebx` for `JSET64_REG`; the two registers are uninitialized for `JSET32_REG`.

One way to implement `JSET32_REG` is to emit a bitwise `and` of `eax` and `ecx`, followed by a conditional jump if the result is non-zero (i.e., the `zf` bit in the `eflags` register is clear). But the JIT emits extra `and` and `or` instructions that also use `edx` and `ebx`, which are uninitialized for `JSET32_REG`, incorrectly modifying `eflags`. The bug was not caught by the BPF selftests suite because none of the tests "polluted" `edx` and `ebx` with values that would cause the behavior to change. We fixed the bug by moving the last two `EMIT2` statements under a condition that `is_jmp64` is true.

There are other bugs in the excerpt: when `DST` is mapped to x86 registers and not spilled on the stack (i.e., `dstk` is false), the emitted instructions incorrectly clobber the registers, while the semantics of the BPF instructions requires `DST` not to change. We fixed the bugs by loading `DST` to `eax` and `ecx`, regardless of whether `DST` is on the stack.

Subtle instruction encoding. Below is an encoding bug in the x86-32 JIT for the BPF `LDXB` instruction, which loads a byte from memory. As its semantics requires the result to be zero-extended to 64 bits, the JIT attempts to emit "`mov dst_hi,0`" to clear the upper 32 bits, using the following C code:

```
EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);
```

Notice that `EMIT3` emits 3 bytes, but a correct "`mov dst_hi,0`" expects 6 bytes: the opcode `0xC7`, the ModR/M byte formed by `add_1reg(0xC0, dst_hi)`, followed by 4 bytes of zeros as the immediate. The consequence is not merely an incorrect `mov`: it also "swallows" 3 bytes from the next instruction, breaking the instruction stream and altering the meaning of the subsequent instructions. We fixed the bug by emitting "`xor dst_hi,dst_hi`" instead, which is also shorter (2 bytes).

4.3.3 Summary

Compared to the bugs in *classic* BPF JITs [34, 175], those in today's BPF JITs are more sophisticated due to the increased power of the BPF virtual machine. On the other hand, architecture-independent checks for BPF programs such as division by zero are now performed by the BPF checker, eliminating the need for the JITs to consider such cases.

While the Arm and RISC-V JITs emit instructions using well-defined macros (e.g., Figure 4.2) or functions (e.g., Figure 4.3), the x86 JITs directly emits raw bytes (e.g., Figure 4.4), partly due to the lack of a uniform instruction format on x86. Jitterbug therefore needs to model the semantics of their target architectures precisely; for x86, this means reasoning at the level of raw instruction bytes.

4.4 Specification

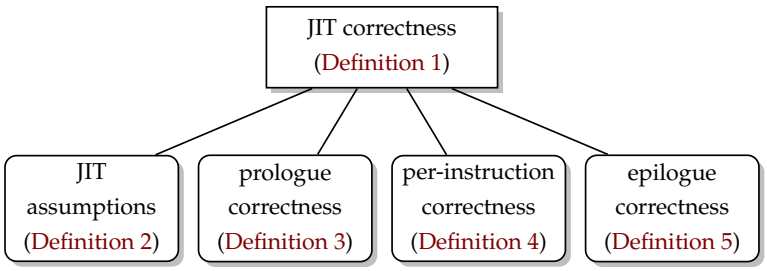

FIGURE 4.5: Jitterbug's stepwise specification (rounded-corner boxes) implies JIT correctness, shown by Theorem 1.

Jitterbug aims to rule out subtle bugs in BPF JITs through a formal specification, which is the focus of this section.

We begin with an intuitive description of what it means for a JIT to be correct. At a high level, running the machine code emitted by a JIT for a given source program should be equivalent to running a BPF interpreter with that source program. For example, both should compute the same return value and invoke the same kernel functions with the same arguments; any deviation indicates a bug. Jitterbug captures this intuition as a JIT correctness specification (subsection 4.4.1).

Specifications like this are usually proved by induction, and the key to carrying out the proof is finding the right inductive invariant—a property preserved by the JIT translation of each individual source instruction. Inspired by the structure of the existing BPF JITs in the Linux kernel, Jitterbug introduces a *stepwise* specification that serves as our inductive invariant. As shown in Figure 4.5, this specification consists of a set of properties satisfied by individual translation steps, such as the generation of machine code for a single BPF instruction. Using the Lean theorem prover [115], we prove that any JIT that satisfies the stepwise specification implies our intuitive notion of correctness. This proof serves as the metatheory for Jitterbug (subsection 4.4.2). The stepwise specification itself is proved automatically for each JIT.

To illustrate how to apply the stepwise specification to prevent bugs, we use the BPF JIT for RV32 as an example. We also analyze alternative JIT implementations to demonstrate the generality of the specification (subsection 4.4.3).

We end this section with a discussion of the limitations of Jitterbug's specification and how it relates to prior compiler correctness specifications (subsection 4.4.4).

4.4.1 JIT correctness

Formalizing JIT correctness requires formalizing the behavior of the JIT, source BPF programs, and target machine programs, as follows.

First, we model a JIT as a function `JITCompile`. It takes a source program $code_S$ and JIT context ctx as input, and returns either a target program $code_T$ on success, denoted as $\texttt{JITCompile}(code_S, ctx) = code_T$; or $\perp$, indicating compilation error. Both source and target programs are represented as partial maps from addresses to instructions; some addresses may be unmapped. We define $code \subseteq code'$ to mean that any address that maps to some instruction in $code$ maps to the same instruction in $code'$.

The JIT context ctx is an implementation-defined data structure. It usually contains compiler configurations (e.g., the base address of the target program allocated by the kernel, denoted by $ctx[\texttt{base}]$) and analysis results of the source program, which are used by the JIT for code generation. We assume that the JIT context is *well-formed* with respect to the source program; this assumption is captured using a predicate $\texttt{wf}(code_S, ctx)$ specified by JIT developers. For example, one may specify that $ctx[\texttt{base}]$ is properly aligned.

Next, we model the execution of both source and target programs as abstract machines, described by a set of states Σ and a state transition function `step`. Given a state $\sigma \in \Sigma$, we write $\sigma[\cdot]$ to refer to a specific component of the state. For example, $\sigma[\texttt{pc}]$ is the value of the program counter.

The `step` function takes as input a state σ, a program $code$, and an oracle denoted by nd. The oracle nd is an infinite sequence of nondeterministically chosen bytes, which are used for modeling external interactions with the kernel (e.g., values loaded from BPF

maps or returned by calls to kernel functions). Given these inputs, the `step` function produces the next state and a trace of externally visible *events* generated by executing the instruction at the program counter, $code[\sigma[\texttt{pc}]]$. The execution gets *stuck* if it triggers undefined behavior (e.g., the address $\sigma[\texttt{pc}]$ is unmapped in *code*). As shorthands, we write $\langle \sigma, code, nd \rangle \Longrightarrow \langle \sigma', tr \rangle$ to mean $\texttt{step}(\sigma, code, nd) = \langle \sigma', tr \rangle$, and $\langle \sigma, code, nd \rangle \Longrightarrow^* \langle \sigma', tr \rangle$ to mean that state σ' is reachable from zero or more applications of `step` starting from state σ, with concatenated trace tr.

The set of events is specific to each machine. For example, consider the BPF machine in Jitterbug. It defines the following events: $\texttt{load}(addr, val)$, $\texttt{store}(addr, val)$, $\texttt{call}(addr, args, val)$, `atomic_begin`, and `atomic_end`. It models each memory load as returning a fresh value provided by the oracle and producing a `load` event in the trace, since BPF maps may be modified outside the execution of a BPF program (subsection 4.3.1). Each `step` may produce zero or more events. For example, the execution of `XADD32` (32-bit atomic exchange-and-add) produces `atomic_begin`, `load`, `store`, and `atomic_end`.

The model assumes read-only code, which prohibits JITs that produce self-modifying code [118]. It also assumes that the execution of a program is deterministic [97, §2.1], since the next state is uniquely determined by the current state, code, and oracle. Both assumptions match the BPF JITs in Linux.

In order to reason about the start and end of execution, each machine defines two predicates:

- $\texttt{initial}(x, ctx, \sigma)$, where σ is an initial state for input x and JIT context ctx; and
- $\texttt{final}(\sigma', v)$, where σ' is a final state with return value v.

Recall that the JIT considers only *safe* source programs. For example, the Linux kernel rejects BPF programs that the BPF checker deems unsafe (subsection 4.3.1). We capture this guarantee with a predicate $\texttt{safe}(code)$, which specifies that executing *code* always reaches a final state (i.e., the execution terminates without triggering any undefined behavior):

$$\begin{aligned} &\forall\ x, \sigma, nd.\ \texttt{initial}(x, ctx, \sigma) \Rightarrow \\ &\quad \exists\ \sigma', tr, v.\ \langle \sigma, code, nd \rangle \Longrightarrow^* \langle \sigma', tr \rangle \wedge \texttt{final}(\sigma', v). \end{aligned}$$

In addition, since a target program generated by the JIT runs within the kernel, it must behave like a regular function and preserve the corresponding calling convention: for example, stack pointer and callee-saved registers must hold the same values before and after the execution. We capture these requirements in the *architectural safety* predicate $\mathcal{A}(\sigma_T, \sigma'_T)$, which constrains the initial and final values of all preserved target registers r to be the same, i.e., $\sigma_T[r] = \sigma'_T[r]$.

Using our model, we define JIT correctness as follows.

Definition 1 (JIT correctness)**.** A JIT is correct if for any safe source program $code_S$, well-formed JIT context ctx, and target program $code_T$ generated by the JIT such that $\texttt{safe}(code_S) \wedge \texttt{wf}(code_S, ctx) \wedge \texttt{JITCompile}(code_S, ctx) = code_T$, the following two conditions hold:

1. The execution of source program $code_S$ and that of target program $code_T$ produce the same trace and return value.

$$\begin{aligned}&\forall\ x, \sigma_S, \sigma_T, nd, tr, v.\\ &\ \texttt{initial}_S(x, ctx, \sigma_S) \wedge \texttt{initial}_T(x, ctx, \sigma_T) \Rightarrow\\ &\ \Big(\big(\exists\ \sigma'_S.\ \langle \sigma_S, code_S, nd\rangle \Longrightarrow^* \langle \sigma'_S, tr\rangle \wedge \texttt{final}_S(\sigma'_S, v)\big) \leftrightarrow\\ &\ \ \big(\exists\ \sigma'_T.\ \langle \sigma_T, code_T, nd\rangle \Longrightarrow^* \langle \sigma'_T, tr\rangle \wedge \texttt{final}_T(\sigma'_T, v)\big)\Big).\end{aligned}$$

2. Any final state reachable by executing target program $code_T$ satisfies architectural safety.

$$\begin{aligned}&\forall\ x, \sigma_T, \sigma'_T, nd, tr, v.\ \texttt{initial}_T(x, ctx, \sigma_T) \wedge \texttt{final}_T(\sigma'_T, v) \wedge\\ &\langle \sigma_T, code_T, nd\rangle \Longrightarrow^* \langle \sigma'_T, tr\rangle \rightarrow \mathcal{A}(\sigma_T, \sigma'_T).\end{aligned}$$

The first property can be viewed as a *bisimulation* between source and target machines [96, §2]: the JIT produces a target program that preserves the behavior of the source program, and any behavior of the target program is permitted by the source program. Additionally, given that the source program is safe, this property implies that the target program produced by the JIT is safe (i.e., terminates without undefined behavior). The second property further requires the target program to correctly save and restore the corresponding architectural state. Both guarantees are critical for in-kernel execution.

4.4.2 Stepwise specification

Given Definition 1, our goal is to devise a stepwise specification (i.e., an inductive invariant) that both implies JIT correctness and is amenable to automated verification. We achieve this goal by imposing structure on the JIT compilation process so that we can reason about the correctness of individual compilation steps, as follows.

Inspired by the existing BPF JITs in the Linux kernel, we suppose that the JIT generates a target program in a *per-instruction* fashion. Specifically, the target program consists of machine instructions for the prologue, each source instruction, and the epilogue (Figure 4.1). We do not assume any particular code layout. For example, one may produce the target program sequentially:

```
code_T = EmitPrologue(ctx)
for i in [0, |code_S| - 1]:
    code_T += EmitInstruction(ctx, i, code_S[i])
code_T += EmitEpilogue(ctx)
```

We formalize our assumptions about the JIT next.

Definition 2 (JIT assumptions)**.** We assume that for any safe source program $code_S$, well-formed JIT context ctx, and target program $code_T$ produced by a JIT such that $\texttt{safe}(code_S) \wedge \texttt{wf}(code_S, ctx) \wedge \texttt{JITCompile}(code_S, ctx) = code_T$, the target program $code_T$ contains the machine instructions produced by each translation step:

- $\exists\ p.\ \texttt{EmitPrologue}(ctx) = p \wedge p \subseteq code_T$.
- $\forall\ i, insn.\ code_S[i] = insn \Rightarrow$
  $\exists\ p.\ \texttt{EmitInstruction}(ctx, i, insn) = p \wedge p \subseteq code_T$.
- $\exists\ p.\ \texttt{EmitEpilogue}(ctx) = p \wedge p \subseteq code_T$.

With these assumptions, the stepwise specification boils down to the correctness of each translation step: `EmitPrologue`, `EmitInstruction`, and `EmitEpilogue`. Jitterbug allows developers to provide two relations as invariants maintained by their JIT implementations:

- $\sigma_S \sim_{ctx} \sigma_T$ relates source state σ_S and target state σ_T with respect to JIT context ctx.

For example, it may specify that the value of a BPF register in σ_S is equal to that of the machine register the JIT uses to realize the BPF register in σ_T.

- $\mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T)$ relates initial target state σ_{T_0} and non-final target state σ_T with respect to JIT context ctx.

For example, the prologue usually saves callee-saved registers to a designated memory region; $\mathcal{I}_{ctx}$ may specify that the values of callee-saved registers in σ_{T_0} are equal to those in that region in σ_T.

Below we describe the correctness definition for each translation step. We denote the empty trace as ϵ.

Definition 3 (Prologue correctness)**.** A JIT emits a correct prologue if executing the prologue results in a target state that establishes the invariants, and produces an empty trace:

$$
\begin{aligned}
&\forall\ code_S, ctx, p, x, \sigma_S, \sigma_T, nd.\ \texttt{wf}(code_S, ctx) \wedge \\
&\quad \texttt{EmitPrologue}(ctx) = p \wedge \\
&\quad \texttt{initial}_S(x, ctx, \sigma_S) \wedge \texttt{initial}_T(x, ctx, \sigma_T) \Rightarrow \\
&\quad \exists\ \sigma'_T.\ \langle \sigma_T, p, nd \rangle \Longrightarrow^* \langle \sigma'_T, \epsilon \rangle \wedge (\sigma_S \sim_{ctx} \sigma'_T) \wedge \mathcal{I}_{ctx}(\sigma_T, \sigma'_T).
\end{aligned}
$$

Definition 4 (Per-instruction correctness)**.** A JIT emits correct target instructions for a given source instruction if executing the emitted instructions results in a target state that preserves the invariants, and produces the same trace as executing the source instruction:

$$\begin{aligned}
&\forall\ code_S, ctx, i, insn, p, \sigma_S, \sigma_T, \sigma_{T_0}, nd, tr.\ \texttt{wf}(code_S, ctx) \wedge \\
&\quad code_S[i] = insn \wedge \sigma_S[\texttt{pc}] = i \wedge \\
&\quad \texttt{EmitInstruction}(ctx, i, insn) = p \wedge \\
&\quad \langle \sigma_S, code_S, nd \rangle \Longrightarrow \langle \sigma_S', tr \rangle \wedge (\sigma_S \sim_{ctx} \sigma_T) \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T) \Rightarrow \\
&\quad \exists\ \sigma_T'.\ \langle \sigma_T, p, nd \rangle \Longrightarrow^* \langle \sigma_T', tr \rangle \wedge (\sigma_S' \sim_{ctx} \sigma_T') \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T').
\end{aligned}$$

Definition 5 (Epilogue correctness)**.** A JIT emits a correct epilogue if executing the epilogue results in a final target state that satisfies architectural safety, and produces the same return value as in the source final state and an empty trace:

$$\begin{aligned}
&\forall\ code_S, ctx, p, \sigma_S, v, \sigma_T, \sigma_{T_0}, nd.\ \texttt{wf}(code_S, ctx) \wedge \\
&\quad \texttt{EmitEpilogue}(ctx) = p \wedge \\
&\quad \texttt{final}_S(\sigma_S, v) \wedge (\sigma_S \sim_{ctx} \sigma_T) \wedge \mathcal{I}_{ctx}(\sigma_{T_0}, \sigma_T) \Rightarrow \\
&\quad \exists\ \sigma_T'.\ \langle \sigma_T, p, nd \rangle \Longrightarrow^* \langle \sigma_T', \epsilon \rangle \wedge \texttt{final}_T(\sigma_T', v) \wedge \mathcal{A}(\sigma_{T_0}, \sigma_T').
\end{aligned}$$

Together, these three properties imply JIT correctness given the JIT assumptions. We prove the following theorem in Lean:

Theorem 1 (Stepwise soundness)**.**

JIT assumptions $\wedge$ prologue correctness $\wedge$ per-instruction correctness $\wedge$ epilogue correctness $\Rightarrow$ JIT correctness.

With Theorem 1 as a metatheory, Jitterbug proves the correctness of a JIT implementation by proving the properties in Definitions 3, 4, and 5 via automated verification (see section 4.5). The JIT context well-formedness `wf` and assumptions are assumed to be correct and trusted. The invariants ($\sim_{ctx}$ and $\mathcal{I}_{ctx}$) are untrusted: if incorrect invariants are provided, verification fails.

4.4.3 Applying the stepwise specification

The stepwise specification is parameterized by assumptions (well-formedness of JIT context `wf`) and invariants ($\sim_{ctx}$ and $\mathcal{I}_{ctx}$), which reflect how JIT developers intend to establish correctness. We illustrate how to apply the stepwise specification to the BPF JIT for RV32 by specifying the assumptions and invariants regarding registers, program counters, and memory. We also describe how one may specify them for the alternative JIT implementations we have considered.

Figure 4.6 shows the design of the BPF JIT for RV32. The upper half denotes the BPF state, including registers (`R0`–`R10`, `AX`), counters (tail-call counter `TCC` and program

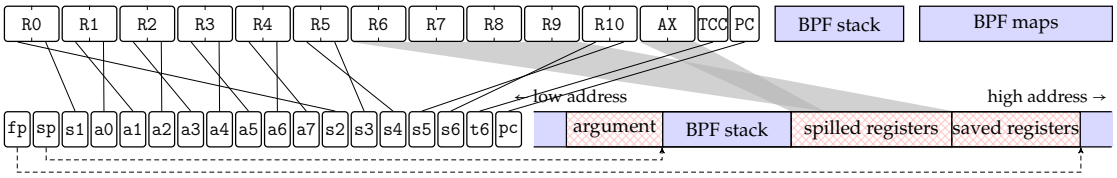

FIGURE 4.6: Mapping from BPF state (upper half) to RV32 state (lower half). Rounded-corner boxes denote registers and rectangular boxes denote memory. Shaded regions are memory accessible by BPF programs and crosshatched regions for internal use.

counter `PC`), a stack memory region, and maps of shared data (subsection 4.3.1). The lower half denotes the RV32 state, including registers (`fp`, `sp`, `a0`–`a7`, `s1`–`s6`, `t6`; those not mapped to BPF registers are omitted), a machine program counter `pc`, and memory.

Registers. Since BPF registers are 64-bit and RV32 is a 32-bit architecture, the JIT realizes each BPF register using either a pair of RV32 registers (e.g., `R1` using `a0` and `a1`) or 64 bits in the "spilled registers" memory region (e.g., `R6`). This register mapping is static and pre-determined, eliminating the need for register allocation at compilation time. Other BPF JITs in the Linux kernel use similar register mappings.

The register mapping is handcrafted to achieve good performance. For instance, recall that BPF designates `R1`–`R5` to pass function-call arguments, while the RISC-V calling convention uses `a0`–`a7`, plus the stack if needed [47]. To minimize register save and restore, the JIT realizes `R1`–`R4` using `a0`–`a7`. For `R5`, the JIT emits instructions to push the corresponding `s3`, `s4` to the "argument" memory region before the call.

To specify the relation $\sigma_S \sim_{ctx} \sigma_T$ between source and target states, let $\varphi_{\text{reg}}(ctx, \sigma_T, r)$ denote the value stored at the target location(s) to which a BPF register r is mapped (e.g., `R1` mapped to `a0`, `a1`) with respect to JIT context ctx. A strawman approach is to require a strict equivalence: $\sigma_S[r] = \varphi_{\text{reg}}(ctx, \sigma_T, r)$ for every BPF register r. With this relation, the stepwise specification would require that if every BPF register and the mapped locations contain equivalent values initially, their values remain equivalent after executing a BPF instruction and the emitted machine instructions, respectively. One such example is the *partial* specification used by the BPF bug finder in Serval [122, §7]; the specification is partial because it does not support reasoning about control flow (e.g., program counters) or memory and cannot be used to prove JIT correctness.

While it is useful for finding bugs, the strawman relation is too restrictive for verification. First, if a BPF program does not use a certain register, it should be safe for the JIT to skip emitting code for initializing the corresponding target locations, but doing so violates the strict equivalence. Second, the relation is difficult to establish in the presence of calls. To see why, consider the BPF register `R1`, which is *not* preserved across a BPF `CALL` instruction (subsection 4.3.1). `R1` is thus considered *uninitialized* after the call as per

the BPF semantics (the BPF checker ensures that `R1` will be written to before any further use). On the other hand, `R1` is mapped to `a0`, `a1`, both of which hold the return value after the call as per the RISC-V calling convention (the JIT emits instructions to further copy their values to `s1`, `s2` to match the BPF calling convention for `R0`). Therefore, `R1` and the corresponding `a0`, `a1` do not hold equivalent values after the call, which violates the strict equivalence.

To relax the strict equivalence and give the JIT more freedom regarding uninitialized BPF registers, we augment the state of the BPF machine with an `initialized` set, which represents the set of registers that are initialized at this point; the set is updated based on the semantics of each BPF instruction. For example, `DST` is added to the set after "`MOV64_IMM DST,IMM`," as it is written to by the instruction. Similarly, `R1`–`R5` are removed from the set after `CALL`, as they are not preserved across the call and become uninitialized. In doing so, it suffices to require equivalence $\sigma_S[r] = \varphi_{\text{reg}}(ctx, \sigma_T, r)$ for every BPF register $r \in \sigma_S[\texttt{initialized}]$, effectively excluding uninitialized ones.

Program counters. Let $\varphi_{\text{pc}}(ctx, i)$ denote the target address to which the i-th BPF instruction is mapped in JIT context ctx. This is useful for a JIT to implement the compilation of jump instructions. It also allows us to relate program counters in BPF and machine states as an invariant $\varphi_{\text{pc}}(ctx, \sigma_S[\texttt{pc}]) = \sigma_T[\texttt{pc}]$.

To define φ_{pc}, one simple approach is to require the JIT to emit a *fixed* number of machine instructions for each BPF instruction (e.g., by padding with NOPs) [61]. In this case, we have $\varphi_{\text{pc}}(ctx, i) = ctx[\texttt{base}] + i \times N$, where $ctx[\texttt{base}]$ is the starting address of the emitted machine instructions determined by JIT context and N is a pre-determined number of machine instructions large enough to compile any BPF instruction. This is simple to specify and implement, but the emitted code wastes space and CPU cycles.

A more efficient approach is to emit a variable number of machine instructions for each BPF instruction. For example, the BPF JITs in the Linux kernel maintain an *offset table* in the JIT context to map each BPF instruction index to an offset into the emitted code; in this case $\varphi_{\text{pc}}(ctx, i)$ is defined by simply consulting the offset table. The JITs construct the offset table by repeating the compilation process until the table converges, or fail if an upper bound on the number of iterations is reached (e.g., 16 in the BPF JIT for RV32).

For flexibility, we choose not to specify how to construct the offset table in the JIT context. Instead, we specify the property a valid JIT context should satisfy. A key observation is that such JITs emit *consecutive* blocks of machine instructions, one block for each BPF instruction. As a result, the difference between the target addresses for a BPF instruction $code_S[i]$ and its successor $code_S[i+1]$ must be equal to the number of bytes emitted for $code_S[i]$. We capture the observation using the well-formedness predicate `wf`

over source program $code_S$ and JIT context ctx for any i-th BPF instruction:

$$\texttt{EmitInstruction}(ctx, i, code_S[i]) = p \Rightarrow$$
$$|p| = \varphi_{\text{pc}}(ctx, i+1) - \varphi_{\text{pc}}(ctx, i).$$

Here $|p|$ denotes the length of machine instructions p (in bytes). This allows for both NOP-padding and the more sophisticated JIT implementations such as those in the Linux kernel. Note that this is an assumption on the validity of the JIT context, which does *not* rule out bugs in the construction of the offset table (see subsection 4.4.4). A JIT may validate the offset table by checking that this predicate holds at compilation time.

Memory. One approach to relating the memory state of source and target machines, denoted by $\sigma_S[\texttt{mem}]$ and $\sigma_T[\texttt{mem}]$, respectively, is to require $\sigma_S[\texttt{mem}](a) = \sigma_T[\texttt{mem}](a)$ for every address a [118], where memory is modeled as a map from addresses to values. But this approach assumes a closed system (see section 4.7 for such a JIT) and does not fit BPF. For example, both user processes and other BPF programs may concurrently modify memory to which a BPF program has access; therefore, consecutive loads from the same address in the BPF program may return different values. A further complication is that BPF does not specify a memory consistency model (subsection 4.3.1), effectively assuming that of the underlying architecture.

We observe that a BPF JIT does not need to reason about the behavior of concurrent memory accesses [32, 102]. Instead, the goal is to faithfully translate BPF memory accesses to ones in the target machine, which is a simpler task. Based on this observation, we employ a hybrid approach to specify the invariants for BPF JITs using traces and maps, as follows.

Each target machine models memory as consisting of two disjoint parts, one corresponding to shared memory and the other for internal use (Figure 4.6). The memory layout used by a JIT determines which target addresses are shared and which are internal. The internal memory is simply a map from addresses to values, since it is private to each execution and the effects are not externally visible. The shared memory captures memory-related effects using events in a trace (subsection 4.4.1). Since the BPF machine adopts the memory model of the underlying architecture, Jitterbug relates the traces of the BPF and target machines by using the same memory model for both; i.e., the BPF and target traces are drawn from the same set of all possible memory events. Given this correspondence, it suffices to require the traces produced by the BPF and target machines to be identical.

The requirement of having identical traces suffices for the BPF JITs. One exception is that older versions of the BPF JIT for Arm64 use Arm's exclusive access instructions in a busy loop [5, §B2], which violates the requirement. Newer versions of the JIT have

switched to using atomic instructions, which satisfies the requirement. We decide not to relax the requirement of having identical traces to keep the specification simple.

4.4.4 Discussion and limitations

Jitterbug's JIT correctness (Definition 1), especially the use of traces, is inspired by the specification of CompCert [96]. Jitterbug's specification differs in the following ways. First, in-kernel execution imposes stricter requirements on the source program (e.g., determinism, termination, and absence of undefined behavior), allowing us to prove stronger properties. Second, Jitterbug uses fine-grained models of target architectures to precisely reason about low-level state (e.g., program counter and stack pointer), whereas CompCert uses a more abstract semantics for assembly [116, §5] and relies on a separate assembler and linker. Third, the per-instruction compilation process of such JITs enables us to develop a stepwise specification amenable to automated verification.

Jitterbug trusts the correctness of the assumptions (subsection 4.4.2). Therefore, it cannot catch bugs in the JIT context (e.g., offset-table construction) or layout of the target program. We manually examine the existing BPF JIT correctness bugs in the Linux kernel (subsection 4.3.2), and determine that out of the 82 bugs, the specification can catch all but two bugs, both in offset-table construction. This shows the effectiveness of the specification.

Jitterbug's specification permits "null" JIT implementations that fail on all source programs; we use existing test suites (e.g., the BPF selftests) to assess the feature completeness of JITs. It focuses on the JIT and cannot rule out bugs in the BPF checker, memory management for code images, or how the kernel uses the JIT. It does not model the instruction cache or memory permissions, relying on the kernel to correctly flush the cache and set up permissions. It does not provide any guarantees against microarchitectural timing channels [85, 60].

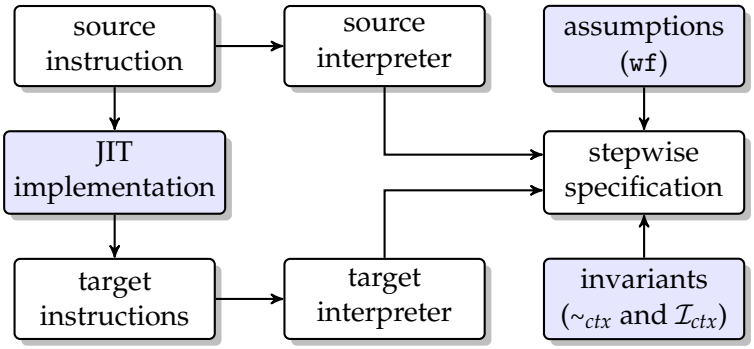

FIGURE 4.7: Jitterbug's verification pipeline. Shaded boxes denote inputs provided by JIT developers.

4.5 Proving JIT correctness

Jitterbug extends automated verification to JIT correctness, a form of compiler correctness tailored for in-kernel execution. This section describes how Jitterbug achieves the automation.

As shown in Figure 4.7, Jitterbug provides the stepwise specification and the executable semantics (i.e., interpreters) for BPF and various architectures. It asks JIT developers for a JIT implementation, and assumptions and invariants regarding the implementation. All the inputs are written in the Rosette language (the JIT is written in the DSL described in section 4.6).

Jitterbug builds on Serval for automated verification [122]. It invokes Rosette to reduce all the inputs to symbolic constraints via symbolic evaluation, and an SMT solver to check the satisfiability of these constraints. For symbolic evaluation to terminate, the JIT implementation and the execution of both interpreters must be free of unbounded loops [169]. The BPF JITs satisfy this requirement.

Below we highlight three key challenges in automated verification of JITs and how Jitterbug addresses these challenges.

Instantiation of existential quantifiers. To prove the stepwise specification, Jitterbug has to construct *some* execution of target instructions emitted by the JIT and show that the execution exhibits the same behavior as that of a source instruction. Automating the construction is challenging.

To see why, consider the specification for per-instruction correctness (subsection 4.4.2). Letting $\vec{x}$ stand for the universally quantified variables in Definition 4, the specification says that the target machine executes some finite number of steps, k, to produce a state σ_T' that satisfies the inductive invariant P. Making the number of steps k explicit, we can write the correctness formula as $\forall \vec{x}.\ \exists k.\ P(\vec{x}, k)$, or equivalently, $\exists f.\ \forall \vec{x}.\ P(\vec{x}, f(\vec{x}))$,

where f is a Skolem function that computes the right k for each combination of the variables $\vec{x}$. The verification problem that Jitterbug solves therefore involves constructing f. In other words, Jitterbug must determine the number of steps to run symbolic evaluation with emitted instructions, and this value $f(\vec{x})$ may depend on the source program, JIT context, source and target states, etc.

In a restricted setting where the JIT emits straight-line code without any branches, $f(\vec{x})$ is simply the number of emitted instructions. The BPF bug finder in Serval and synthesis-based superoptimizers [129] all assume this setting and use the corresponding basic realization of f. But Jitterbug considers JITs that can emit code with branches, and when executing such code, the target machine can take a different number of steps depending on the input state. This rules out the straightforward realization of f that counts the number of emitted instructions.

To illustrate the challenge of computing f in our setting, consider the instructions emitted by the RV32 JIT for the BPF instruction "`JNE64_REG DST,SRC,OFF`" (jump to offset `OFF` if the values in `DST` and `SRC` differ). The JIT may emit different blocks of RV32 instructions, conditioned on whether it spills the registers (requiring `lw` to load from stack) or the offset requires a far jump (`jal` or `auipc+jalr`). Figure 4.8 shows three examples of these blocks and the $f(\vec{x})$ values for executing them, which vary depending on the register state and instructions. In general, constructing f requires human insight [182, 111], so Jitterbug allows JIT developers to provide a manually constructed f. In practice, however, Jitterbug can automate the construction of f for BPF JITs as follows.

To compute f, Jitterbug requires the target interpreter to maintain the symbolic program counter in the form `base` + *offset*, where `base` is the (symbolic) starting address of instructions. Maintaining this form is straightforward for most instructions. For instructions with subtler semantics, the interpreter achieves this by rewriting the program counter via *symbolic optimization* [122, 131]. For example, RISC-V's `jalr` sets the least-significant bit of the program counter to zero (subsection 4.3.2), causing it to take the form (`base` + *offset*) `&` *mask*. The interpreter rewrites this expression by dropping the mask and checking that the resulting expression is equivalent (i.e., that the program counter is properly aligned).

Given a program counter of the form `base` + *offset*, Jitterbug provides a reusable procedure for constructing f through symbolic evaluation. It extracts the offset from the program counter expression and applies a simple rule: stop symbolic evaluation either if the offset is concrete but leaves the block of emitted instructions, or if the offset becomes symbolic. The intuition is that branching with a symbolic offset likely leaves the block, because the JIT generally produces such branching instructions by consulting the offset table in the JIT context (subsection 4.4.3), which is considered symbolic for verification. Internal branching tends to have a concrete offset, for which symbolic evaluation continues.

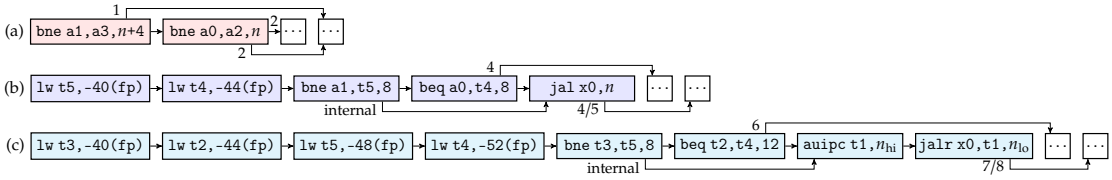

FIGURE 4.8: Emitted RV32 instruction blocks for BPF's `JNE64_REG` with registers (a) `R1`, `R2`; (b) `R1`, `R6`; and (c) `R6`, `R7` and with an offset of different ranges. `R1` and `R2` are realized using RV32 registers, and `R6` and `R7` are spilled on the stack (Figure 4.6). Straight and elbow arrows denote falling through and branching, respectively; those leaving the blocks are labeled with values of $f(\vec{x})$.

This procedure guesses an f for verifying that the target machine reaches a desired state after taking $f(\vec{x})$ steps. It does not guarantee to find the right f if one exists, though it is sufficient for all the JITs we have studied and works well in practice. The procedure is untrusted: choosing a wrong f causes the target machine to either get stuck or enter a state that violates the inductive invariant, but it does not cause an erroneous JIT implementation to pass verification.

Symbolic evaluation of symbolic instructions. As shown in Figure 4.8, the JIT may emit different blocks of target instructions for a given source instruction opcode. When Jitterbug symbolically evaluates a JIT implementation, it produces a symbolic representation of all of these instruction blocks. This representation takes the form of *symbolic* instructions that may contain symbolic values in register and immediate fields. To verify the JIT, Jitterbug must then evaluate the target interpreter on both a symbolic input state *and* a symbolic program. This is in contrast to prior work on verifying systems code such as Serval, where the input state is symbolic but the program itself is concrete (e.g., all register and immediate fields are concrete bytes). Reasoning about symbolic programs both magnifies existing challenges to scaling verification and creates new ones. We discuss one example of each.

The first challenge is path explosion. While common to all tools based on symbolic evaluation, this problem becomes exponentially worse in the presence of symbolic code. For example, the BPF JIT for RV64 compiles `LD64_IMM` to a variable number of instructions to load a 64-bit immediate in chunks, selecting each instruction based on the chunk value and destination register. This amounts to reasoning about a total of 2,181 types of blocks of RV64 instructions for downstream stages, out of which 307 are feasible, applied to all possible input instructions (roughly, 2^{64}). *Symbolic execution* [41, 80], which explores individual paths separately, is thus not a good fit for this verification pipeline.

Jitterbug instead adopts Rosette's strategy for symbolic evaluation [168, §4] to merge the program state at each control-flow join, but it forces a split on every possible (concrete) opcode of symbolic instructions. The intuition is that both the JIT and interpreters

tend to handle each opcode separately; splitting on the opcode enables opportunities for concrete evaluation. This strategy works well in practice: it avoids path explosion and leads to easier-to-solve constraints.

The second challenge is that Jitterbug interpreters, unlike those in Serval, must be designed to work on both symbolic state and symbolic instructions. Failing to do so both causes state explosion and produces constraints difficult for SMT solving. For example, the interpreters in Serval represent the CPU state using a vector of bitvectors (one bitvector per register), and encode accessing register r_i as indexing into the vector using integer i. This is suitable for concrete instructions, where i is concrete and the generated constraints are restricted to the theory of bitvectors. But with symbolic instructions, a symbolic register index i causes symbolic evaluation to produce constraints that also use the theory of (mathematical) integers. Mixing integers and bitvectors is expensive for solving and can lead to verification bottlenecks [72, §3].

We develop interpreters that account for symbolic instructions and thus can work with Jitterbug. For example, we carefully avoid integers in instruction semantics to restrict resulting constraints to the theories of equality with uninterpreted functions and bitvectors, a decidable fragment of first-order logic. Additionally, recall that the BPF JITs for x86 emit raw bytes (subsection 4.3.2) and thus require a decoder for verification. We implement an x86 decoder that works on symbolic bytes. The development process is guided by using symbolic profiling to identify verification performance bottlenecks [22] and applying symbolic optimization to fine-tune symbolic evaluation [122, §4].

Axiomatization of expensive SMT operations. Both BPF and machine interpreters provide arithmetic instructions for multiplication, division, and remainder. Reasoning about these operations is expensive for SMT solvers [86, 10], and has been a source of timeouts in verification practice [58, 68].

To avoid such expensive reasoning, Jitterbug takes a standard *axiomatization* approach [87, §3.2] by replacing these bitvector operations with uninterpreted functions $\texttt{mul}_n$, $\texttt{div}_n$, and $\texttt{rem}_n$ ($n \in \{32, 64\}$) in instruction semantics. For example, the BPF JIT for RV32 translates BPF's `DIV32_REG` using RISC-V's `divu`; both instructions are encoded using the uninterpreted function $\texttt{div}_{32}$ (with variations for handling division by zero). Proving the correctness of this translation does *not* require the semantics of division, thereby scaling verification.

This approach is less general than using SMT's built-in bitvector operations, because it ignores the semantics of these operations and might reject valid JIT implementations. For example, the JIT may reorder the operands of a multiplication in emitted instructions; for target architectures lacking native instructions for remainder or 64-bit multiplication, the JIT may emit instructions that emulate the behavior. Proving such a JIT correct requires additional properties about the operations. Jitterbug captures these

properties using the following axioms, which are sufficient to verify all the JITs we have studied:

- commutativity of `mul`: $\texttt{mul}_n(x,y) = \texttt{mul}_n(y,x)$;
- remainder: $\texttt{rem}_n(x,y) = x - \texttt{mul}_n(\texttt{div}_n(x,y),y)$;
- commutativity of `mulhu`: $\texttt{mulhu}_n(x,y) = \texttt{mulhu}_n(y,x)$; and
- decomposition of $\texttt{mul}_{64}$:
  $\texttt{mul}_{64}(x,y) = (\texttt{mulhu}_{32}(x_{\text{lo}},y_{\text{lo}}) + \texttt{mul}_{32}(x_{\text{hi}},y_{\text{lo}}) + \texttt{mul}_{32}(x_{\text{lo}},y_{\text{hi}})) \oplus \texttt{mul}_{32}(x_{\text{lo}},y_{\text{lo}})$.

Here x and y are bitvectors; subscripts "lo" and "hi" denote the lower and upper 32 bits of a bitvector, respectively; $\oplus$ denotes bitvector concatenation; and `mulhu` is an auxiliary uninterpreted function for modeling the upper bits of a product. For example, x86's 32-bit unsigned multiplication instruction stores a 64-bit product split across registers `edx` and `eax`; the x86 interpreter models their values using $\texttt{mulhu}_{32}$ and $\texttt{mul}_{32}$, respectively.

These axioms are shared by the verification of the BPF JITs across architectures. As a sanity check, we formalize and manually prove them using Lean [115].

4.6 Implementing a JIT

DSL. Figure 4.9 shows an excerpt of the BPF JIT for RV32, written in the Jitterbug DSL. The DSL is implemented as a Rosette library and reflects a structured subset of C: booleans, (machine) integers, array accesses ("`@`"), as well as conditional and switch statements. This subset is minimal and sufficient to support the development of the BPF JIT for RV32. It does not support address-of, dereference, or unstructured control flow constructs (e.g., `goto` or fallthrough in `switch`).

Jitterbug extracts the final C code from JIT fragments written in the DSL, a code template (including glue code not covered by the DSL), and a type mapping (not shown here; both array accesses to `bpf2rv32` and calls to `bpf_get_reg64` return a value of type "`const s8 *`"). Jitterbug does not perform any type checking for the C code.

Using the DSL simplifies verification by avoiding the need to model the C semantics. One can also "escape" from the DSL to use the full Rosette language, though in that case Jitterbug is unable to perform C code extraction; we leverage this to simplify porting the existing BPF JITs from C to Jitterbug.

```
(func (emit_alu_r64 dst src ctx op)
  (var [tmp1 (@ bpf2rv32 TMP_REG_1)]
       [tmp2 (@ bpf2rv32 TMP_REG_2)]
       [rd   (bpf_get_reg64 dst tmp1 ctx)]
       [rs   (bpf_get_reg64 src tmp2 ctx)])
  (switch op
    [(BPF_ADD)
      (cond
        [(equal? rd rs)
          (emit (rv_srli RV_REG_T0 (lo rd) 31) ctx)
          (emit (rv_slli (hi rd) (hi rd) 1) ctx)
          (emit (rv_or (hi rd) RV_REG_T0 (hi rd)) ctx)
          (emit (rv_slli (lo rd) (lo rd) 1) ctx)]
        [else
          (emit (rv_add (lo rd) (lo rd) (lo rs)) ctx)
          (emit (rv_sltu RV_REG_T0 (lo rd) (lo rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) (hi rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) RV_REG_T0) ctx)])]
    ...))
```

(a) JIT implementation written in the DSL.

```
void emit_alu_r64(const s8 *dst, const s8 *src,
                  struct rv_jit_context *ctx, const u8 op)
{
// clang-format on
@|emit_alu_r64|
// clang-format off
}
```

(b) C code template, where `@|...|` expands to generated code.

```
void emit_alu_r64(const s8 *dst, const s8 *src,
                  struct rv_jit_context *ctx, const u8 op)
{
  const s8 *tmp1 = bpf2rv32[TMP_REG_1];
  const s8 *tmp2 = bpf2rv32[TMP_REG_2];
  const s8 *rd = bpf_get_reg64(dst, tmp1, ctx);
  const s8 *rs = bpf_get_reg64(src, tmp2, ctx);

  switch (op) {
  case BPF_ADD:
    if (rd == rs) {
      emit(rv_srli(RV_REG_T0, lo(rd), 31), ctx);
      emit(rv_slli(hi(rd), hi(rd), 1), ctx);
      emit(rv_or(hi(rd), RV_REG_T0, hi(rd)), ctx);
      emit(rv_slli(lo(rd), lo(rd), 1), ctx);
    } else {
      emit(rv_add(lo(rd), lo(rd), lo(rs)), ctx);
      emit(rv_sltu(RV_REG_T0, lo(rd), lo(rs)), ctx);
      emit(rv_add(hi(rd), hi(rd), hi(rs)), ctx);
      emit(rv_add(hi(rd), hi(rd), RV_REG_T0), ctx);
    }
    break;
  ...
}
```

(c) Final (extracted) JIT implementation in C.

FIGURE 4.9: Excerpt of the BPF JIT for RV32 for compiling the "`ADD64_REG DST,SRC`" instruction.

Synthesis. As another application of Jitterbug's specification and verification, we use Rosette's support for program synthesis to optimize the BPF JIT for RV32 [106]. We do so by synthesizing JIT fragments written in (a subset of) the DSL, where each fragment takes as input a BPF instruction with a given opcode (e.g., `ADD64_REG`) and emits a short sequence of RV32 instructions with equivalent behavior. We use the standard approach of writing program sketches [159, 23] to compactly define a space of JIT fragments for

compiling `ALU` instructions. The synthesizer searches this space for the shortest fragment that satisfies per-instruction correctness (Definition 4), according to the Jitterbug verifier.

Using this approach, we found two JIT fragments better than our manual implementation for compiling `ADD64_REG` (Figure 4.9) and `SUB64_REG`. In each case, the synthesized fragment emitted four instructions, whereas our manual implementation emitted five. We adopted the synthesized fragments in the JIT.

4.7 Experience

Component (in Rosette)	Lines of code
Jitterbug framework	1,825
BPF interpreter	471
Arm32 interpreter	1,265
Arm64 interpreter	1,166
RISC-V interpreter (32- and 64-bit)	1,571
x86 interpreter (32- and 64-bit)	2,299

FIGURE 4.10: Line counts of Jitterbug's components.

	JIT impl.		Spec.	Per-opcode verification time			
	C	DSL		Min	Max	Mean	Median
RV32	1,964	1,420	336	16	401	73	55
RV64	1,862	1,225	284	4	7,542	116	24
Arm32	1,620	839	192	23	925	130	99
Arm64	1,025	653	163	4	110	26	23
x86-32	1,683	1,074	185	24	488	122	109
x86-64	1,382	644	182	5	170	33	27

FIGURE 4.11: Line counts and per-opcode verification time (in seconds) of the BPF JITs for six architectures.

Figure 4.10 shows the code size of the Jitterbug framework and the interpreters for verifying JITs, all written in Rosette. We wrote the interpreters in an idiomatic way [68, §3.3], adding instructions as needed. We borrowed part of the BPF and RV64 semantics from Serval [122], but rewrote the interpreters to support symbolic instructions (section 4.5); we wrote the others from scratch. We developed the metatheory for JIT correctness and the bitvector axiomatization using 1,492 lines of Lean code.

The primary application of Jitterbug is a new BPF JIT for RV32, which we wrote in the DSL, proved against the stepwise specification, and extracted to a C implementation. To validate the generality of Jitterbug, we ported the existing BPF JITs for RV64, Arm32, Arm64, x86-32, and x86-64 in the Linux kernel to Jitterbug for verification. Each

port was line-by-line transcription from C to the DSL (and Rosette), emitting the same instructions as the original JIT. These ports did not cover the support for legacy instruction sets (e.g., those lacking atomic instructions mentioned in subsection 4.4.3), compiling `TAIL_CALL`, or optimizing register saving.

Figure 4.11 lists the line counts for each BPF JIT. The specification effort comprises writing assumptions and invariants for the implementation (section 4.5). Since Jitterbug performs verification for each source instruction opcode individually, we measured the per-opcode verification time, using an Intel Core i7-7700K CPU at 4.5 GHz, with Boolector 3.2.1 as the SMT solver [124]. Verification time across the JITs depends on many factors (e.g., the JIT implementation or solver), though architectural differences are a contributing factor. For example, the most time-consuming case is verifying the JIT for RV64 with BPF's `LD64_IMM` (loading a 64-bit immediate), which emits 307 types of blocks of RISC-V instructions; the JIT for x86-64 emits six types for the same opcode.

Below we describe our experience using Jitterbug for the BPF JITs and a previously verified JIT for a stack machine [118].

The BPF JIT for RV32. We chose to implement a BPF JIT for RV32 because there was no such JIT in the Linux kernel. The development took five iterations of code reviews.

The first two iterations occurred in June 2019. We sent an initial implementation to kernel developers to gather feedback and gauge interest. We wrote the implementation in Rosette and manually translated it to C, and was unverified. The feedback was positive, with suggestions to add support for eliminating zero extensions [174], an optimization BPF had just introduced.

We submitted the third implementation in February 2020. It was switched to using the DSL (section 4.6), which was less prone to errors in manual translation. It passed the BPF selftests suite, and was verified against an early version of the specification. One of the suggestions from kernel developers was to factor out the common code to be shared among the JITs, such as the per-instruction structure (subsection 4.4.2). We addressed the suggestions in the next two iterations, after which the JIT was accepted into the Linux kernel (see subsection A.1.1).

Throughout this process, we refined both the specification and the implementation. The early version of the specification missed two bugs that were also missed by testing. The first bug was an off-by-one error for `TAIL_CALL`: the emitted instructions limited the `TCC` (tail-call counter) to 32, rather than the correct value 33. The second bug was that the JIT did not maintain 16-byte alignment of the stack as per the calling convention. We found the two bugs once we completed the specification.

Automated verification supported this development process in two ways. First, it minimized the proof burden for developing the JIT, which must be feature-complete for deployability. Second, it enabled us to catch up with new features being introduced

(e.g., support for eliminating zero extensions) and address code reviews by kernel developers in a timely manner.

Code review. As listed in subsection A.1.2, we found 16 new bugs in the existing BPF JITs, wrote patches that fix these bugs, and verified the fixed code. In addition, we found two new bugs in the Arm64 instruction encoding library, a core component shared by BPF and other kernel subsystems (e.g., KVM). We wrote new test cases to be included in the BPF selftests suite. This is useful for catching similar bugs across the JITs, as various "bots" are running selftests continuously for the Linux kernel. Finding subtle bugs in well-tested code shows the effectiveness of the specification and verification.

The main effort for porting and verifying these JITs was in writing the target interpreters for Jitterbug. Verifying the JITs for Arm32 and Arm64 took one week each. Verifying the JITs for x86-32 and x86-64 took three weeks in total, due to the complexity of the x86 interpreter (e.g., instruction decoding). Translating C code to Rosette was mechanical and straightforward, though mistakes in manual translation might hide bugs; extending Jitterbug to work on C code is future work. For specification, we adopted the assumptions and invariants for the JIT for RV32 and adjusted them accordingly.

In our experience, automated verification is key to rapid code review using formal methods. As an example, in December 2019, the developer of the BPF JIT for RV64 submitted patches to add support for far jumps. We ported the patches to Jitterbug and verified their correctness within days of the patch submission. We reported the verification results to kernel developers; the patches were accepted with our review.

Optimization. Another advantage of verification is that it enables developing complex optimizations by providing a high degree of confidence in their correctness. As listed in subsection A.1.3, we developed 12 patches optimizing the existing BPF JITs. Like code review, we verified the correctness of these optimizations by manually translating the C code to Rosette.

One of the optimizations adds support for RISC-V compressed instruction-set extension (RVC) to the BPF JIT for RV64. RVC improves code density and reduces instruction cache misses by adding short 2-byte instructions for common operations [177, §5], but it poses a challenge to verification: the JIT may choose either base (4-byte) or RVC (2-byte) for emitting each instruction, depending on the immediate value or registers. This leads to an exponential increase in the number of paths in the JIT, emitted instructions, and machine state (e.g., variable code lengths causing the program counter to take different values). Developing and verifying this optimization took approximately 3 weeks, following the proof strategy described in section 4.5 to scale verification.

Beyond BPF JITs. While Jitterbug focuses on the BPF JITs, we also applied it to a JIT for a stack machine to x86-32. We ported the "version 1" JIT described by Myreen [118] to the Jitterbug DSL and extracted it to C code; the port emitted the same x86-32 instructions and was able to run the example as in the paper (Jitterbug does not support the "version 2" JIT that emits self-modifying code). For specification, we excluded registers from the invariants, since the stack machine had no registers; and modeled memory as a map from addresses to values without using traces, since the stack machine had no shared memory (subsection 4.4.3). For verification, we wrote an interpreter for the stack machine and reused the x86 interpreter provided by Jitterbug. This took one day.

Jitterbug reported two bugs in the JIT implementation: the offsets for two conditional jump instructions are given as 5 in the original paper, but we concluded that the correct value should be 8. We fixed the offsets and verification succeeded. We believe that both are typos in the paper, as our (fixed) JIT is consistent with the paper's HOL4 code and proof.

4.8 Reflection

Our work on Jitterbug was inspired by an earlier effort, started in 2015, to use the Coq theorem prover to develop a verified BPF JIT for x86-64. We chose to implement the JIT itself in x86-64 so as to minimize the trusted computing base. In hindsight, this was a mistake: doing so required reasoning about low-level machine state for both the compiler and emitted code, which hindered the completion of the proof; and the JIT implementation was impractical to audit and deploy due to the lack of C code and the optimizations seen in the Linux kernel. We suspended this effort two years later, in 2017.

Our interest in BPF JITs was revived with the development of symbolic profiling [22] and optimization [122, 131], which together demonstrated a systematic approach for scaling automated verification of low-level code. As an experiment, we wrote a bug finder for BPF JITs in Serval, which checked for strict equivalence of registers (subsection 4.4.3) over straight-line instructions (section 4.5). It enabled us to find 15 bugs regarding `ALU` instructions in two BPF JITs, although it was insufficient for verification or finding the bugs described in subsection 4.3.2 due to the lack of a correctness specification and proof strategy (e.g., support for symbolic instructions).

For Jitterbug, we spent most of our effort devising a specification of JIT correctness that is general enough to cover a broad range of in-kernel JITs (e.g., without requiring padding emitted instructions), expressive enough to catch real bugs, and amenable to automated verification. We found the use of the Lean theorem prover valuable for navigating this trade-off, developing several iterations of the stepwise specification and a proof that implies the correctness of compiling entire programs. It also improved our confidence in the axiomatization of bitvector operations.

A key lesson from Jitterbug is that deciding what *not* to verify is as important as deciding what to verify. For instance, while ideally we would write and verify the BPF JIT for RV32 in C directly, the use of the DSL enabled us to fine-tune symbolic evaluation, which was critical for scaling verification. If we could not scale verification to JITs written in the DSL, verifying JITs written in C would surely be out of reach. Inspired by seL4 [152] and Ironclad [68], we bridged the resulting gap through validation, separately verifying that the instruction encoding functions in C emitted the same bytes as their original DSL code.

Chapter 5

Conclusion and future directions

This dissertation has demonstrated how to lower the effort required to write automated verifiers for systems code, and how to retrofit systems for automated verification. 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. With Jitterbug, we extend Serval to develop automated verifiers for BPF JITs, a critical and rapidly evolving component in the Linux kernel.

There are still challenges to overcome before there can be widespread adoption of automated verification in practice; these challenges present promising directions for future investigation. First, many production systems are continually evolving, which may require changing specifications or needing to develop new verification techniques. Keeping verification in sync with such systems as they change will likely require better integration of automated verification tools into the workflows of system developers without requiring that they also be experts in formal verification. Second, automated verification works only on systems with specifications expressible in first-order logic and interfaces that can be implemented without unbounded loops. While this covers an interesting class of systems like security monitors, many systems do not meet these criteria. To scale verification to such systems, future tools may need to combine different verification techniques to benefit from the generality of techniques like interactive theorem proving as well as the low developer burden of automated verification. We hope that the ideas from this dissertation will help to guide future advances in automated verification techniques and drive wider adoption to prevent bugs in this critical layer of software.

Appendix A

Appendix

A.1 Patches to the Linux kernel developed using Jitterbug

The following tables list the upstreamed patches to the Linux kernel that we have developed using Jitterbug.

A.1.1 Development of the BPF JIT for RV32

Commit	Architecture	Description
5f316b65e99f	RV32	Add RV32G eBPF JIT
ca6cb5447cec	RV32	Factor common RISC-V JIT code
745abfaa9eaf	RV32	Fix tail call count off by one in RV32 BPF JIT
91f658587a96	RV32	Fix stack layout of JITed code on RV32

A.1.2 Bug fixes and new test cases

Commit	Architecture	Description
bb9562cf5c67	Arm32	Fix bugs with ALU64 RSH, ARSH BPF_K shift by 0
4178417cc535	Arm32	Fix offset overflow for BPF_MEM BPF_DW
579d1b3faa37	Arm64	Fix two bugs in encoding 32-bit logical immediates
1e692f09e091	RV64	Clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh
489553dd13a8	RV64	Fix offset range checking for auipc+jalr on RV64
6fa632e719ee	x86-32	Fix bug with ALU64 LSH, RSH, ARSH BPF_K shift by 0
68a8357ec15b	x86-32	Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0
80f1f8503635	x86-32	Fix bug with JMP32 JSET BPF_X checking upper bits
5fa9a98fb103	x86-32	Fix incorrect encoding in BPF_LDX zero-extension
50fe7ebb6475	x86-32	Fix clobbering of dst for BPF_JSET
aee194b14dd2	x86-64	Fix encoding for lower 8-bit registers in BPF_STX BPF_B
d2b6c3ab70db	–	Add test for BPF_STX BPF_B storing R10
93e5fbb18cec	–	Add test for JMP32 JSET BPF_X with upper bits set
ac8786c72eba	–	Add tests for shifts by zero

A.1.3 Optimizations for existing BPF JITs

Commit	Architecture	Description
cf48db69bdfa	Arm32	Optimize ALU64 ARSH X using orrpl conditional instruction
c648c9c7429e	Arm32	Optimize ALU ARSH K using asr immediate instruction
fd49591cb49b	Arm64	Optimize AND,OR,XOR,JSET BPF_K using arm64 logical immediates
fd868f148189	Arm64	Optimize ADD,SUB,JMP BPF_K using arm64 add/sub immediates
46dd3d7d287b	RV64	Enable zext optimization for more RV64G ALU ops
0224b2acea0f	RV64	Enable missing verifier_zext optimizations on RV64
21a099abb765	RV64	Optimize FROM_LE using verifier_zext on RV64
ca349a6a104e	RV64	Optimize BPF_JMP BPF_K when imm == 0 on RV64
073ca6a0369e	RV64	Optimize BPF_JSET BPF_K using andi on RV64
bfabff3cb0fe	RV64	Modify JIT ctx to support compressed instructions
804ec72c68c8	RV64	Add encodings for compressed instructions
18a4d8c97b84	RV64	Use compressed instructions in the rv64 JIT

A.2 Bug-fixing commits in BPF JITs in the Linux kernel

The following table lists bug-fixing commits in the BPF JITs in the Linux kernel for Arm32, Arm64, RV64, x86-32, and x86-64 from May 2014 through April 2020. The superscripts *J* and *S* mark those for fixing bugs found using Jitterbug and the BPF bug finder in Serval, respectively.

Commit	Architecture	Year	Description
`ALU`:			
`bb9562cf5c67`^{J}	Arm32	2020	Fix bugs with alu64 rsh, arsh bpf_k shift by 0
`14e589ff4aa3`	Arm64	2015	Fix mod-by-zero case
`251599e1d690`	Arm64	2015	Fix div-by-zero case
`d63903bbc30c`	Arm64	2015	Fix endianness conversion bugs
`1e4df6b72081`	Arm64	2015	Fix signedness bug in loading 64-bit immediate
`1e692f09e091`^{S}	RV64	2019	Clear high 32 bits for alu32 add/sub/neg/lsh/rsh/arsh
`fe121ee531d1`	RV64	2019	Clear target register high 32-bits for and/or/xor on alu32
`6fa632e719ee`^{S}	x86-32	2019	Fix bug with alu64 lsh, rsh, arsh bpf_k shift by 0
`68a8357ec15b`^{S}	x86-32	2019	Fix bug with alu64 lsh, rsh, arsh bpf_x shift by 0
`b9aa0b35d878`	x86-32	2019	Fix bug for bpf_alu64 | bpf_neg
`343f845b3759`	x86-64	2015	Fix from_be16 and from_le16/32 instructions
`JMP`:			
`2b589a7e2bd3`	Arm32	2018	Correct check_imm24
`ddc665a4bb4b`	Arm64	2017	Fix jit branch offset related to ldimm64
`8eee539ddea0`	Arm64	2015	Fix out-of-bounds read in bpf2a64_offset()
`50fe7ebb6475`^{J}	x86-32	2020	Fix clobbering of dst for bpf_jset
`80f1f8503635`^{J}	x86-32	2020	Fix bug with jmp32 jset bpf_x checking upper bits
`711aef1bbf88`	x86-32	2019	Fix bug for bpf_jmp | bpf_jsgt, bpf_jsle, bpf_jslt, bpf_jsge
`7c2e988f400e`	x86-64	2019	Fix x64 jit code generation for jmp to 1st insn
`MEM`:			
`4178417cc535`^{J}	Arm32	2020	Fix offset overflow for bpf_mem bpf_dw
`ec19e02b343d`	Arm32	2018	Fix ldx instructions
`8968c67a82ab`	Arm64	2019	Remove prefetch insn in xadd mapping
`7005cade1bdb`	Arm64	2017	Use separate register for state in stxr
`5ca1ca01fae1`	x86-32	2020	Fix logic error in bpf_ldx zero-extension
`5fa9a98fb103`^{J}	x86-32	2020	Fix incorrect encoding in bpf_ldx zero-extension
`aee194b14dd2`^{J}	x86-64	2020	Fix encoding for lower 8-bit registers in bpf_stx bpf_b
`CALL`:			
`8c11ea5ce13d`	Arm64	2018	Fix getting subprog addr from aux for calls
`489553dd13a8`^{J}	RV64	2020	Fix offset range checking for auipc+jalr on rv64
`TAIL_CALL` and `EXIT`:			
`02088d9b392f`	Arm32	2018	Fix register saving
`f4483f2cc1fd`	Arm32	2018	Fix tail call jumps
`51c9fbb1b146`	Arm64	2014	Lift restriction on last instruction
`16338a9b3ac3`	Arm64	2018	Fix out of bounds access in tail call
`a2284d912bfc`	Arm64	2018	Fix stack_depth tracking in combination with tail calls
`d8b54110ee94`	Arm64	2017	Fix faulty emission of map access in tail calls
`96bc4432f5ad`	RV64	2019	Limit to 33 tail calls
`769e0de6475e`	x86-64	2014	Fix epilogue generation for ebpf programs
`90caccdd8cc0`	x86-64	2017	Fix bpf_tail_call() x64 jit
`2482abb93ebf`	x86-64	2015	Fix general protection fault when tail call is invoked
Prologue and epilogue:			
`d1220efd2348`	Arm32	2018	Fix stack alignment
`f1003b787c00`	RV64	2019	Fix broken bpf tail calls
`9e4e5b5c8666`	x86-32	2018	Fix regression caused by commit 24dea04767e6
`fe8d9571dc50`	x86-64	2019	Fix stack layout of jited bpf code

Bibliography

[1] Eyad Alkassar et al. "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, Aug. 2010, pp. 71–85.

[2] Sidney Amani et al. "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, Apr. 2016, pp. 175–188.

[3] Nadav Amit and Michael Wei. "The Design and Implementation of Hyperupcalls". In: *Proceedings of the 2018 USENIX Annual Technical Conference*. Boston, MA, July 2018, pp. 97–111.

[4] Nadav Amit et al. "Virtual CPU Validation". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 311–327.

[5] *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*. Mar. 2020.

[6] Alasdair Armstrong et al. "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, Jan. 2019.

[7] John Aycock. "A Brief History of Just-In-Time". In: *ACM Computing Surveys* 35.2 (June 2003), pp. 97–113.

[8] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Survey* 51.3 (July 2018).

[9] Mike Barnett et al. "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, Nov. 2005, pp. 364–387.

[10] Paul Beame and Vincent Liew. "Towards Verifying Nonlinear Integer Arithmetic". In: *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*. Heidelberg, Germany, July 2017, pp. 238–258.

[11] Adam Belay et al. “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, Oct. 2012, pp. 335–348.

[12] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proceedings of the 2005 USENIX Annual Technical Conference*. Anaheim, CA, Apr. 2005, pp. 41–46.

[13] Brian N. Bershad et al. “Extensibility, Safety and Performance in the SPIN operating system”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, CO, Dec. 1995, pp. 267–284.

[14] William R. Bevier. “Kit: A Study in Operating System Verification”. In: *IEEE Transactions on Software Engineering* 15.11 (Nov. 1989), pp. 1382–1396.

[15] Sven Beyer et al. “Putting it all together – Formal verification of the VAMP”. In: *International Journal on Software Tools for Technology Transfer* 8.4–5 (Aug. 2006), pp. 411–430.

[16] Armin Biere et al. “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, Mar. 1999, pp. 193–207.

[17] Ashish Bijlani and Umakishore Ramachandran. “Extension Framework for File Systems in User space”. In: *Proceedings of the 2019 USENIX Annual Technical Conference*. Renton, WA, July 2019, pp. 121–134.

[18] Dion Blazakis. “Interpreter Exploitation: Pointer Inference and JIT Spraying”. In: *Black Hat DC*. Arlington, VA, Feb. 2010.

[19] Sandrine Blazy and Xavier Leroy. “Mechanized semantics for the Clight subset of the C language”. In: *Journal of Automated Reasoning* 43.3 (Oct. 2009), pp. 263–288.

[20] Barry Bond et al. “Vale: Verifying High-Performance Cryptographic Assembly Code”. In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada, Aug. 2017, pp. 917–934.

[21] Daniel Borkmann. *bpf, x86, arm64: Enable jit by default when not built as always-on*. Commit `81c22041d9f1`. Linux kernel, Dec. 2019.

[22] James Bornholt and Emina Torlak. “Finding Code That Explodes Under Symbolic Evaluation”. In: *Proceedings of the 33rd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Boston, MA, Nov. 2018.

[23] James Bornholt et al. "Optimizing Synthesis with Metasketches". In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, Jan. 2016, pp. 775–788.

[24] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. "The Boyer-Moore Theorem Prover and Its Interactive Enhancement". In: *Computers and Mathematics with Applications* 29.2 (Jan. 1995), pp. 27–62.

[25] Jo Van Bulck et al. "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, Aug. 2018, pp. 991–1008.

[26] Cristian Cadar. "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, Aug. 2015, pp. 906–909.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "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, Dec. 2008, pp. 209–224.

[28] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90.

[29] Cristian Cadar et al. "EXE: Automatically Generating Inputs of Death". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. Alexandria, VA, Oct. 2006, pp. 322–335.

[30] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. "Dynamic Instrumentation of Production Systems". In: *Proceedings of the 2004 USENIX Annual Technical Conference*. Boston, MA, June 2004, pp. 15–28.

[31] Quentin Carbonneaux et al. "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, June 2014, pp. 270–281.

[32] Tej Chajed et al. "Verifying concurrent software using movers in CSPEC". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, Oct. 2018, pp. 307–322.

[33] Craig Chambers and David Ungar. "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language". In: *Proceedings of the 10th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, June 1989, pp. 46–160.

[34] Haogang Chen et al. "Security bugs in embedded interpreters". In: *Proceedings of the 4th Asia-Pacific Workshop on Systems*. 6 pages. Singapore, July 2013.

[35] Haogang Chen et al. "Using Crash Hoare Logic for Certifying the FSCQ File System". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 18–37.

[36] Haogang Chen et al. "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, Oct. 2017, pp. 270–286.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "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, Mar. 2011, pp. 265–278.

[38] Adam Chlipala. "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, Jan. 2015, pp. 609–622.

[39] Maria Christakis and Patrice Godefroid. "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, Jan. 2015, pp. 373–392.

[40] Andrey Chudnov et al. "Continuous formal verification of Amazon s2n". In: *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*. Oxford, United Kingdom, July 2018, pp. 430–446.

[41] Lori A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs". In: *IEEE Transactions on Software Engineering* 2.3 (May 1976), pp. 215–222.

[42] Jonathan Corbet. *BPF at Facebook (and beyond)*. `https://lwn.net/Articles/801871/`. Oct. 2019.

[43] Jonathan Corbet. *Concurrency management in BPF*. `https://lwn.net/Articles/779120/`. Feb. 2019.

[44] Jonathan Corbet. *Post-init read-only memory*. `https://lwn.net/Articles/666550/`. Dec. 2015.

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

[46] David Costanzo, Zhong Shao, and Ronghui Gu. "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, June 2016, pp. 648–664.

[47] Palmer Dabbelt et al. *RISC-V ELF psABI specification*. `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`. Aug. 2020.

[48] David L. Detlefs et al. *Extended Static Checking*. Research Report SRC-RR-159. Compaq Systems Research Center, Dec. 1998.

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

[50] Jake Edge. *A trio of fuzzers*. `https://lwn.net/Articles/705937/`. Nov. 2016.

[51] Dawson R. Engler. "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System". In: *Proceedings of the 17th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadephia, PA, May 1996, pp. 160–170.

[52] Dawson R. Engler and M. Frans Kaashoek. "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation". In: *Proceedings of the 1996 ACM SIGCOMM Conference*. Stanford, CA, Aug. 1996, pp. 53–59.

[53] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, CO, Dec. 1995, pp. 251–266.

[54] Andres Erbsen et al. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises". In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019, pp. 73–90.

[55] Manuel Fähndrich et al. "Language Support for Fast and Reliable Message-based Communication in Singularity OS". In: *Proceedings of the 1st ACM EuroSys Conference*. Leuven, Belgium, Apr. 2006, pp. 177–190.

[56] Matthias Felleisen et al. "A Programmable Programming Language". In: *Communications of the ACM* 61.3 (Mar. 2018), pp. 62–71.

[57] Matthias Felleisen et al. "A Programmable Programming Language". In: *Communications of the ACM* 61.3 (Mar. 2018), pp. 62–71.

[58] Andrew Ferraiuolo et al. "Komodo: Using verification to disentangle secure-enclave hardware from software". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017, pp. 287–305.

[59] Matt Fleming. *A thorough introduction to eBPF*. `https://lwn.net/Articles/740157/`. Dec. 2017.

[60] Qian Ge et al. "Time Protection: The Missing OS Abstraction". In: *Proceedings of the 14th ACM EuroSys Conference*. Dresden, Germany, Mar. 2019.

[61] Jacob Van Geffen et al. "Synthesizing JIT Compilers for In-Kernel DSLs". In: *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV)*. Los Angeles, CA, July 2020, pp. 564–586.

[62] Elazar Gershuni et al. "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Phoenix, AZ, June 2019, pp. 1069–1084.

[63] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing". In: *Communications of the ACM* 55.3 (Mar. 2012), pp. 40–44.

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

[65] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison Wesley, 2020.

[66] Ronghui Gu et al. "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, Nov. 2016, pp. 653–669.

[67] Ronghui Gu et al. "Deep Specifications and Certified Abstraction Layers". In: *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, Jan. 2015, pp. 595–608.

[68] Chris Hawblitzel et al. "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, Oct. 2014, pp. 165–181.

[69] Chris Hawblitzel et al. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 1–17.

[70] Gernot Heiser, Gerwin Klein, and June Andronick. "seL4 in Australia: From Research to Real-World Trustworthy Systems". In: *Communications of the ACM* 63.4 (Apr. 2020), pp. 72–75.

[71] Toke Høiland-Jørgensen et al. “The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel”. In: *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. Heraklion, Greece, Dec. 2018, pp. 54–66.

[72] Jingmei Hu et al. “Trials and Tribulations in Synthesizing Operating Systems”. In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. Huntsville, Ontario, Canada, Oct. 2019, pp. 67–73.

[73] Daniel Jackson and Jeannette Wing. “Lightweight Formal Methods”. In: *IEEE Computer* 29.4 (Apr. 1996), pp. 20–22.

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

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

[76] M. Frans Kaashoek et al. “Application Performance and Flexibility on Exokernel Systems”. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France, Oct. 1997, pp. 52–65.

[77] Daniel Kästner et al. “CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler”. In: *Proceedings of the 9th European Congress Embedded Real-Time Software and Systems*. Toulouse, France, Jan. 2018, pp. 1–9.

[78] David Keppel, Susan J. Eggers, and Robert R. Henry. *A Case for Runtime Code Generation*. Tech. rep. 91-11-04. University of Washington, Nov. 1991.

[79] Jakub Kicinski and Nicolaas Viljoen. “eBPF Hardware Offload to SmartNICs: cls_bpf and XDP”. In: *the 3rd Technical Conference on Linux Networking*. Tokyo, Japan, Oct. 2016.

[80] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394.

[81] Gerwin Klein. “Operating system verification—An overview”. In: *Sādhanā* 34.1 (Feb. 2009), pp. 27–69.

[82] Gerwin Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–70.

[83] Gerwin Klein et al. “Formally Verified Software in the Real World”. In: *Communications of the ACM* 61.10 (Oct. 2018), pp. 68–77.

[84] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, Oct. 2009, pp. 207–220.

[85] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019, pp. 19–37.

[86] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. "Complexity of Fixed-Size Bit-Vector Logics". In: *Theory of Computing Systems* 59.2 (Aug. 2016), pp. 323–376.

[87] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.

[88] Volodymyr Kuznetsov et al. "Efficient State Merging in Symbolic Execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, June 2012, pp. 193–204.

[89] Leslie Lamport. *Computation and State Machines*. Apr. 2008.

[90] Butler W. Lampson and Robert F. Sproull. "An Open Operating System for a Single-User Machine". In: *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, CA, Dec. 1979, pp. 98–105.

[91] Chris Lattner and Vikram Adve. "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, Mar. 2004, pp. 75–86.

[92] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler Validation via Equivalence Modulo Inputs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014, pp. 216–226.

[93] Dayeol Lee et al. *Keystone: A Framework for Architecting TEEs*. `https://arxiv.org/abs/1907.10119`. July 2019.

[94] K. Rustan M. Leino. "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, Apr. 2010, pp. 348–370.

[95] K. Rustan M. Leino and Michał Moskal. "Usable Auto-Active Verification". In: *Workshop on Usable Verification*. Redmond, WA, Nov. 2010.

[96] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446.

[97] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115.

[98] Xavier Leroy et al. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012.

[99] Amit Levy et al. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017, pp. 234–251.

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

[101] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018, pp. 973–990.

[102] Jacob R. Lorch et al. "Armada: Low-Effort Verification of High-Performance Concurrent Program". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, June 2020, pp. 197–210.

[103] Haohui Mai et al. "Verifying Security Invariants in ExpressOS". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, Mar. 2013, pp. 293–304.

[104] Marek Majkowski. *Cloudflare architecture and how BPF eats the world*. `https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/`. May 2019.

[105] Michaël Marcozzi et al. "Compiler Fuzzing: How Much Does It Matter?" In: *Proceedings of the 34th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Athens, Greece, Oct. 2019.

[106] Henry Massalin. "Superoptimizer: A Look at the Smallest Program". In: *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Palo Alto, CA, Oct. 1987, pp. 122–126.

[107] Henry Massalin and Calton Pu. "Threads and Input/Output in the Synthesis Kernel". In: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*. Litchfield Park, AZ, Dec. 1989, pp. 191–201.

[108] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the Winter 1993 USENIX Technical Conference*. San Diego, CA, Jan. 1993, pp. 259–270.

[109] Frank McKeen et al. “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, June 2016.

[110] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. “The Packet Filter: An Efficient Mechanism for User-level Network Code”. In: *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*. Austin, TX, Nov. 1987, pp. 39–51.

[111] J Strother Moore. *Piton: A Verified Assembly Level Language*. Tech. rep. 22. Computational Logic, Inc., Sept. 1988.

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

[113] Leonardo de Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: Introduction and Applications”. In: *Communications of the ACM* 54.9 (Sept. 2011), pp. 69–77.

[114] Leonardo de Moura and Nikolaj Bjørner. “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, Mar. 2008, pp. 337–340.

[115] Leonardo de Moura et al. “The Lean Theorem Prover”. In: *Proceedings of the 25th International Conference on Automated Deduction (CADE)*. Berlin, Germany, Aug. 2015, pp. 378–388.

[116] Eric Mullen et al. “Verified Peephole Optimizations for CompCert”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, June 2016, pp. 448–461.

[117] Toby Murray et al. “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, May 2013, pp. 415–429.

[118] Magnus O. Myreen. “Verified Just-In-Time Compiler on x86”. In: *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, Jan. 2011, pp. 107–118.

[119] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. “Extensible Proof-Producing Compilation”. In: *Proceedings of the 18th International Conference on Compiler Construction*. York, United Kingdom, Mar. 2009, pp. 2–16.

[120] George C. Necula and Peter Lee. "Safe Kernel Extensions Without Run-Time Checking". In: *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Oct. 1996, pp. 229–243.

[121] Luke Nelson et al. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017, pp. 252–269.

[122] Luke Nelson et al. "Scaling symbolic evaluation for automated verification of systems code with Serval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, Oct. 2019, pp. 225–242.

[123] Luke Nelson et al. "Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual conference, Nov. 2020, pp. 41–61.

[124] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 9 (2015), pp. 53–58.

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

[126] Jan Nordholz. "Design of a Symbolically Executable Embedded Hypervisor". In: *Proceedings of the 15th ACM EuroSys Conference*. Heraklion, Greece, Apr. 2020.

[127] Liam O'Connor et al. "Refinement Through Restraint: Bringing Down the Cost of Verification". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Nara, Japan, Sept. 2016, pp. 89–102.

[128] Justin Pettit et al. "Bringing Platform Harmony to VMware NSX". In: *ACM SIGOPS Operating Systems Review* 52.1 (Aug. 2018), pp. 123–128.

[129] Phitchaya Mangpo Phothilimthana et al. "Scaling up Superoptimization". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, Apr. 2016, pp. 297–310.

[130] Rob Pike, Bart N. Locanthi, and John Reiser. "Hardware/Software Trade-offs for Bitmap Graphics on the Blit". In: *Software: Practice and Experience* 15.2 (Feb. 1985), pp. 131–151.

[131] Sorawee Porncharoenwase, James Bornholt, and Emina Torlak. "Fixing Code That Explodes Under Symbolic Evaluation". In: *Proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. New Orleans, LA, Jan. 2020, pp. 44–67.

[132] *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, Oct. 2014.

[133] *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, Nov. 2016.

[134] *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, Oct. 2018.

[135] *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, CO, Dec. 1995.

[136] *Proceedings of the 1996 ACM SIGCOMM Conference*. Stanford, CA, Aug. 1996.

[137] *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, Apr. 2016.

[138] *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015.

[139] *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017.

[140] *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018.

[141] *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Oct. 1996.

[142] *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014.

[143] *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, June 2016.

[144] *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2019.

[145] *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, Jan. 2015.

[146] Jonathan Protzenko et al. "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider". In: *Proceedings of the 41st IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2020, pp. 983–1002.

[147] Alastair Reid. "Who Guards the Guards? Formal Validation of the ARM v8-M Architecture Specification". In: *Proceedings of the 32nd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vancouver, Canada, Oct. 2017.

[148] Dennis M. Ritchie. *An Incomplete History of the QED Text Editor*. `https://www.bell-labs.com/usr/dmr/www/qed.html`. Feb. 2004.

[149] John Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Tech. rep. CSL-92-02. SRI International, Dec. 1992.

[150] Margo I. Seltzer et al. "Dealing With Disaster: Surviving Misbehaved Kernel Extensions". In: *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Oct. 1996, pp. 213–227.

[151] Arvind Seshadri et al. "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, Oct. 2007, pp. 335–350.

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

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

[154] Helgi Sigurbjarnarson et al. "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, Oct. 2018, pp. 287–306.

[155] Helgi Sigurbjarnarson et al. "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, Nov. 2016, pp. 1–16.

[156] KP Singh. *MAC and Audit policy using eBPF (KRSI)*. `https://lkml.org/lkml/2020/3/28/479`. Mar. 2020.

[157] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Montreal, Canada, Aug. 2008, pp. 28–32.

[158] Louis Sobel. *eJitk: Extending Jitk to eBPF*. `https://css.csail.mit.edu/6.888/2015/papers/ejitk_sobel.pdf`. May 2015.

[159] Armando Solar-Lezama et al. "Combinatorial Sketching for Finite Programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, Oct. 2006, pp. 404–415.

[160] Venkatesh Srinivasan and Thomas Reps. "Partial Evaluation of Machine Code". In: *Proceedings of the 30th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, Oct. 2015, pp. 860–879.

[161] Alexei Starovoitov. *bpf: introduce BPF_JIT_ALWAYS_ON config*. Commit `290af86629b2`. Linux kernel, Jan. 2018.

[162] Alexei Starovoitov. *net: filter: rework/optimize internal BPF interpreter's instruction set*. Commit `bd4cf0ed331a`. Linux kernel, Mar. 2014.

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

[164] Chuck P. Thacker et al. *Alto: A personal computer*. Tech. rep. CSL-79-11. Xerox Palo Alto Research Center, Aug. 1979.

[165] The Clang Team. *UndefinedBehaviorSanitizer*. 2019. URL: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[166] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*. July 2020. DOI: `10.5281/zenodo.4021912`. URL: `https://doi.org/10.5281/zenodo.4021912`.

[167] Ken Thompson. "Regular Expression Search Algorithm". In: *Communications of the ACM* 11.6 (June 1968), pp. 419–422.

[168] Emina Torlak and Rastislav Bodik. "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, June 2014, pp. 530–541.

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

[170] Lourival Vieira Neto et al. "Scriptable Operating Systems with Lua". In: *Proceedings of the 10th Dynamic Languages Symposium*. Portland, OR, Oct. 2014, pp. 2–10.

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

[172] Robert Wahbe et al. "Efficient Software-Based Fault Isolation". In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. Asheville, NC, Dec. 1993, pp. 203–216.

[173] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. "ASHs: Application-Specific Handlers for High-Performance Messaging". In: *Proceedings of the 1996 ACM SIGCOMM Conference*. Stanford, CA, Aug. 1996, pp. 40–52.

[174] Jiong Wang. *bpf: eliminate zero extensions for sub-register writes*. `https://lore.kernel.org/bpf/1558736728-7229-1-git-send-email-jiong.wang@netronome.com/`. May 2019.

[175] Xi Wang et al. “Jitk: A Trustworthy In-Kernel Interpreter Infrastructure”. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, Oct. 2014, pp. 33–47.

[176] Xi Wang et al. “Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior”. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, Nov. 2013, pp. 260–275.

[177] Andrew Waterman. “Design of the RISC-V Instruction Set Architecture”. PhD thesis. University of California, Berkeley, Jan. 2016.

[178] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. RISC-V Foundation, Dec. 2019.

[179] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation. June 2019.

[180] Jean Yang and Chris Hawblitzel. “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, June 2010, pp. 99–110.

[181] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, June 2011, pp. 283–294.

[182] William D. Young. *A Verified Code Generator for a Subset of Gypsy*. Tech. rep. 33. Computational Logic, Inc., Oct. 1988.

[183] Fengzhe Zhang et al. “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, Oct. 2011, pp. 203–216.

[184] Jean-Karim Zinzindohoué et al. “HACL*: A Verified Modern Cryptographic Library”. In: *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, Oct. 2017.