Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, Zachary Tatlock
MIT CSAIL and University of Washington

Abstract

Modern operating systems run multiple interpreters in the kernel, which enable user-space applications to add new functionality or specialize system policies. The correctness of such interpreters is critical to the overall system security: bugs in interpreters could allow adversaries to compromise user-space applications and even the kernel.

Jitk is a new infrastructure for building in-kernel interpreters that guarantee *functional correctness* as they compile user-space policies down to native instructions for execution in the kernel. To demonstrate Jitk, we implement two interpreters in the Linux kernel, BPF and INET-DIAG, which are used for network and system call filtering and socket monitoring, respectively. To help application developers write correct filters, we introduce a high-level rule language, along with a proof that Jitk correctly translates high-level rules all the way to native machine code, and demonstrate that this language can be integrated into OpenSSH with tens of lines of code. We built a prototype of Jitk on top of the CompCert verified compiler and integrated it into the Linux kernel. Experimental results show that Jitk is practical, fast, and trustworthy.

1 Introduction

Many operating systems allow user-space applications to customize and extend the kernel by downloading user-specified code into the kernel [1, 24]. One well-known example is the BSD Packet Filter (BPF) architecture [48]. With BPF, applications specify which packets they are interested in by downloading a filter program into the kernel that decides whether a packet should be dropped or forwarded to the application. For portability and safety, the kernel usually defines a simple, restricted language, and uses an *interpreter* to execute code written in that language (e.g., BPF), rather than directly downloading and executing machine code. Other notable applications of in-kernel interpreters include socket monitoring [40], dynamic tracing [7], power management [32], and system call filtering [20]. Interpreters are also used outside of kernels, such as in Bitcoin's transaction scripting [2].

As an example, consider the Seccomp subsystem [20] in the Linux kernel, which adopts the BPF language to specify what system calls a process can make. Seccomp's overall architecture is shown in Figure 1. At start-up, an application such as OpenSSH submits a BPF filter into the kernel. The kernel invokes the BPF interpreter to run the filter code against every subsequent system call. Based on the result from the interpreter, the kernel decides whether to reject or allow a system call, or kill the process altogether. Figure 2 shows the system call filter used by OpenSSH, written in the BPF language [48]. Even if an adversary later compromises the OpenSSH process, she cannot perform damaging actions, such as modifying files, as the kernel would fail the corresponding system calls, which are disallowed by the installed filter. Many other applications, such as QEMU, Chrome, Firefox, vsftpd, and Tor, secure themselves in a similar fashion.

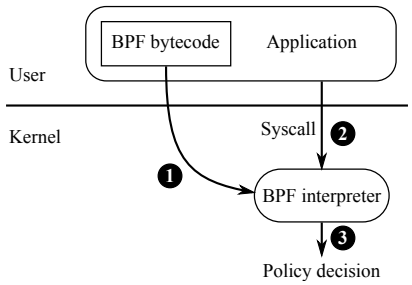

Figure 1: The architecture of the Seccomp system [20] in Linux. Application developers specify their system call policy as a BPF filter (e.g., Figure 2), in bytecode form. At start-up, the user-space application submits the filter to the kernel. The kernel invokes a BPF interpreter to evaluate the program against each subsequent system call, and decides whether to allow or reject it based on the result from the interpreter.

```
      ; load syscall number
      ld   [0]
      ; deny open() with errno = EACCES
      jeq  #SYS_open, L1, L2
L1:   ret  #RET_ERRNO|#EACCES
      ; allow getpid()
L2:   jeq  #SYS_getpid, L3, L4
L3:   ret  #RET_ALLOW
      ; allow gettimeofday()
L4:   jeq  #SYS_gettimeofday, L5, L6
L5:   ret  #RET_ALLOW
L6:   ...
      ; default: kill current process
      ret  #RET_KILL
```

Figure 2: The system call filter used in OpenSSH, in the BSD Packet Filter (BPF) language [48]. It forces the `open` system call to fail with the errno code `EACCES`, allows system calls such as `getpid` and `gettimeofday`, and kills the current process if it invokes other system calls. The `ld` instruction loads the current system call number into the accumulator register; `jeq` n, l_t, l_f is a conditional jump instruction that branches to l_t if the accumulator register is n, and otherwise branches to l_f; and `ret` terminates the filter with a return value.

```
{ default_action = Kill;
  rules = [
    { action = Errno EACCES; syscall = SYS_open };
    { action = Allow; syscall = SYS_getpid };
    { action = Allow; syscall = SYS_gettimeofday };
    ...
  ] }
```

Figure 3: OpenSSH's system call filter from Figure 2, expressed in our higher-level System Call Policy Language (SCPL).

The security of these systems critically relies on both the interpreter and user-supplied code. Since the interpreter resides in kernel-space and has full privileges, bugs in the interpreter can enable the adversary to take control of the entire system [39]. Even if a kernel compromise does not occur, bugs in the interpreter can cause it to produce the wrong result. In Seccomp, this means that the kernel may fail to stop illegal system call invocations, and thereby allow the security of the user-space application to be compromised. Finally, if user-space applications submit incorrect code to start with, such as a BPF filter that lets unintended system calls slip through, the kernel would be unable to enforce the correct policy.

Unfortunately, it is challenging to ensure that both the in-kernel interpreter and the supplied user-specified code are bug-free. First, the interpreter has a complex interface to the external world, leaving a wide range of attack vectors to adversaries who can control either user-specified code (e.g., BPF filters) or input data to the code (e.g., system call invocations), or even both. Second, the interpreter needs to handle many corner cases, such as out-of-bounds memory accesses, jumps to illegal kernel code, arithmetic errors, and infinite loops, which have historically caused problems in many systems (see §3). Third, many in-kernel interpreters employ just-in-time (JIT) compilation to convert code into native machine instructions for better performance [15, 37]; this adds another level of complexity. Fourth, there are few tools that can ensure the correctness of user-specified code.

This paper presents Jitk, a new in-kernel interpreter infrastructure that addresses these challenges through formal verification. Jitk implements a JIT that translates two languages used in the Linux kernel, BPF [48] and INET-DIAG [40], into native code, including x86, ARM, and PowerPC, and proves *functional correctness* of this translation. Jitk guarantees that the resulting native code for in-kernel execution preserves the semantics of the BPF or INET-DIAG code submitted from user space, and that the native code never performs damaging operations such as division by zero or out-of-bounds memory access.

To extend the benefits of functional correctness to user-space applications, Jitk introduces a new high-level specification language called SCPL (System Call Policy Language). Application developers can use SCPL to specify their desired system call policies using *intuitive* rules, such as "allow the `gettimeofday` system call" or "reject the `open` system call with `EACCES`." As an example, Figure 3 shows the SCPL rules that capture the BPF filter used by OpenSSH shown in Figure 2. The hope is that it is less likely for developers to make mistakes in SCPL rules than in manually written BPF filters. Jitk implements a SCPL-to-BPF compiler and a functional correctness proof from these high-level policies to native code.

The code and proof of Jitk were developed using the Coq proof assistant on top of the CompCert framework [42]. We integrated Jitk with the Linux kernel, as a drop-in replacement of its existing Seccomp subsystem. Applications like OpenSSH can run on our system without modifications, with the guarantee of the absence of interpreter bugs described in §3.

Overall, the contributions of this paper are as follows:

- The Jitk infrastructure and approach for building verified in-kernel JIT interpreters.
- A case study of real-world vulnerabilities found in BPF interpreters in several operating systems.
- A formalization of correctness and safety goals for executing user-specified policies in the kernel.
- The Jitk/BPF and Jitk/INET-DIAG verified JITs along with the formal specifications of both languages.
- The SCPL high-level language for specifying system call policy rules, along with a proof of correctness for an SCPL-to-BPF compiler.
- An evaluation of how well Jitk's formal verification prevents the vulnerabilities that have been discovered in bytecode interpreters in real-world kernels.

The rest of the paper is organized as follows. §2 discusses previous related work. §3 presents background information on the kinds of bugs that arise in in-kernel interpreters such as BPF. §4 provides an overview of Jitk's architecture and goals. §5 describes the design and proof. §6 discusses the limitations of Jitk's approach. §7 presents our prototype implementations of Jitk as applied in Seccomp and INET-DIAG in the Linux kernel. §8 evaluates Jitk's security and performance. §9 concludes.

2 Related work

Pioneering work such as seL4 [38, 54], CompCert [42], MinVisor [49], VCC [41], and Myreen's x86 JIT compiler [50] showed the promise of formal verification for building trustworthy, critical software systems, including OS kernels, compilers, and hypervisors. Jitk demonstrates how to apply formal techniques to building systems that download and execute untrusted code in a commodity kernel. Jitk leverages CompCert's compiler infrastructure and machine code semantics; alternatively, Jitk could be built on the Bedrock library [12, 13].

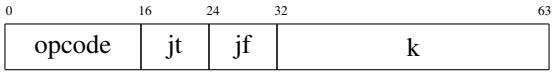

Figure 4: A BPF instruction in bytecode format, with a fixed length of 8 bytes (64 bits). It consists of an opcode, true and false target offsets for conditional jump instructions, and a general field *k*.

There is a rich literature of securing and isolating faulty kernel extension through OS design, such as microkernels [18, 23, 31, 55] and exokernels [24, 34]; through the use of type-safe languages, such as SPIN (Modula-3) [1], Singularity (C#) [33], and Mirage (OCaml) [45]; through software-based fault isolation [58], such as BGI [9], LXFI [46], XFI [25], VINO [53], and SVA [17]; and through proof-carrying code [52]. These techniques focus on memory safety and kernel integrity, by isolating user-specified code from the rest of the kernel, but cannot guarantee functional correctness of the downloaded code.

Testing tools such as EXE [5], KLEE [6], and the Trinity syscall fuzzer [36] are useful for finding bugs in kernel code, and have even been applied to BPF interpreters, but they cannot guarantee bug-free code.

§3 extends Chen et al.'s earlier survey [10] with a case study of a wider range of in-kernel interpreter bugs, and the rest of the paper presents the design and implementation of Jitk that guarantees the absence of these bugs.

Our experience with Jitk suggests that it is feasible to build formally verified JITs in the kernel, on the basis of CompCert. Using a verified compiler like CompCert provides stronger assurance guarantees than some of the alternative proposals, such as integrating the LLVM infrastructure into the kernel [11].

3 Case study

Enforcing a policy in Seccomp, such as the one shown in Figure 2, involves several steps: programmers express the policy to a user-space application, which submits the policy to the kernel, which in turn relays it to an interpreter, which then either purely interprets it or compiles it to machine code for faster execution. This section summarizes representative bugs at each of these steps, using BPF as an example, and discusses the challenges of achieving correctness in this chain. Note that these bugs are general and *not* BPF-specific; they have appeared in other interpreters as well [10].

3.1 Background: the BPF virtual machine

BPF is a general-purpose virtual machine, consisting of:

- a 32-bit accumulator *A*,
- a 32-bit index register *X*,
- a scratch memory *M* for temporary storage (e.g., 64 bytes in the Linux kernel),
- an input packet *P* (the data blob for inspection), and
- an implicit program counter *pc*.

Opcode	Operands	Description
`ld`	[*k*]	$A \leftarrow P[k, .., k+3]$
`ja`	#*k*	$pc \leftarrow pc + k$
`jeq`	#*k*, `jt`, `jf`	$pc \leftarrow pc + (A = k)$? `jt` : `jf`
`div`	#*k*	$A \leftarrow A / k$
`ret`	#*k*	return *k*

Figure 5: Examples of BPF instructions. See the original BPF paper for a complete list [48].

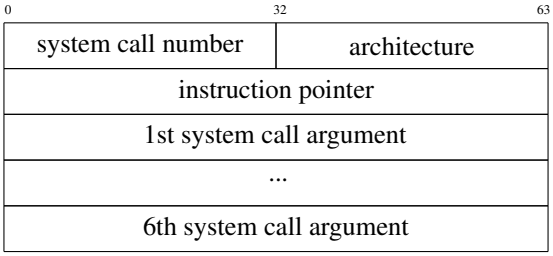

Figure 6: Input to system call filters in the Linux kernel [20], a 64-byte (512-bit) packet. It consists of the current system call number, the architecture, the instruction pointer, and up to six system call arguments.

A BPF filter is a sequence of BPF instructions, each of which has a fixed length of 8 bytes, as shown in Figure 4. It can read the input packet *P*, transfer data among the two registers (*A* and *X*) and the scratch memory *M*, perform arithmetic operations, and terminate with a 32-bit integer return value, which instructs the kernel to take further actions. Figure 5 shows examples of BPF instructions used in this paper; see the original BPF paper for a more complete list [48].

The BPF virtual machine has been successfully applied in different contexts. Its original purpose is to inspect network packets, with the return value indicating the number of bytes to accept. Its applications have gone beyond that [3, 16, 20]. For example, the Seccomp system in the Linux kernel uses BPF for system call filtering: the kernel prepares a 64-byte input packet storing the current system call arguments, as shown in Figure 6, and the return value indicates whether the kernel should fail this system call.

A *well-defined* BPF filter must end with a `ret` instruction; it can jump only forward; and it cannot perform illegal operations such as division by zero, out-of-bounds memory access, or jumping to non-existent instructions. Bugs can arise if an interpreter fails to reject illegal BPF filters, as we will show next.

3.2 Kernel-space bugs

The complex logic for executing BPF filters happens in kernel-space, where bugs can be disastrous for security. Figure 7 lists common errors that have appeared in existing BPF interpreters in Linux and BSD kernels, as detailed next.

Bug description	Examples
Kernel: control-flow errors (§3.2.1)	
jump target off by one	CVE-2014-2889 [39]
jump offset integer overflow	NetBSD PR #3366 [19], OpenBSD cvs bpf_filter.c:r1.13
Kernel: arithmetic errors (§3.2.2)	
incorrect divison-by-zero check	NetBSD PR #43185 [29], OpenBSD cvs bpf_filter.c:r1.21
incorrect reciprocal multiplication	Linux git `aee636c480` [21]
Kernel: memory errors (§3.2.3)	
buffer overflow	NetBSD PR #32198 [27], Linux git `fe15f3f1`, FreeBSD svn 182380
array index integer overflow	NetBSD PR #45751 [51], Linux git `55820ee2`, FreeBSD svn 41588
Kernel: information leak (§3.2.4)	
uninitialized read (scratch memory)	NetBSD PR #45142 [30], CVE-2010-4158 (Linux), CVE-2012-3729 (iOS)
uninitialized read (A & X registers)	Linux git `83d5b7ef99` [56]
Kernel-user interface bugs (§3.3)	
incorrect bytecode encoding/decoding	Linux git `8c482cdc` [4]
User-space bugs (§3.4)	
incorrect translation	tcpdump git `489f459b` [35], libseccomp git `cc063d8d`
incorrect optimization	tcpdump issue #38 [26], tcpdump issue #42

Figure 7: Representative bugs in BPF interpreters.

```
/* x86 code: jcc t_offset; jmp f_offset; ...
   t_offset should be increased by
   (a) 2 bytes (jmp rel8) or
   (b) 5 bytes (jmp rel32) */
jcc = /* conditional jump opcode */
if (filter[i].jf) /* BUG: should be 2 : 5 */
    t_offset += is_near(f_offset) ? 2 : 6;
EMIT_COND_JMP(jcc, t_offset);
if (filter[i].jf)
    EMIT_JMP(f_offset);
```

Figure 8: Incorrect jump target (off by one) in the BPF x86 JIT of the Linux kernel (CVE-2014-2889 [39]). The size of a `jmp` here is either 2 or 5 bytes, not 6.

```
if (BPF_OP(insn->code) == BPF_JA) {
    /* BUG: miss overflow case pc + insn->k < pc */
    if (pc + insn->k >= len)
        return 0;
}
```

Figure 9: Insufficient validation of the jump offset k [19]: given a large k, $pc + k$ will wrap around to a smaller value and bypass the check.

3.2.1 Control-flow errors

JIT interpreters need to correctly translate the control flow of a BPF filter to machine code, which is both delicate and intricate; even a tiny typo can open a new door for kernel exploits. As an example, Figure 8 shows an off-by-one bug in the x86 JIT in the Linux kernel: for a BPF conditional jump, the interpreter emits an x86 conditional jump instruction (for the true case), followed by an unconditional `jmp rel32` (for the false case), which is 5 bytes; the interpreter mistakes it as 6 bytes, and increases the offset of the conditional jump instruction by that wrong value. Consequently, the conditional jump instruction will go one byte past the target instruction, which can be an arbitrary payload controlled by an adversary [39].

```
case BPF_DIV: /* reject A / k where k = 0 */
  /* BUG: should be 0x08; 0x18 is a wrong mask */
  if ((insn->code & 0x18) == BPF_K && insn->k == 0)
      return 0;
```

Figure 10: Incorrect division-by-zero check [29]. The code uses the wrong mask `0x18`, and thus fails to reject BPF code that performs division by zero, which may lead to a kernel crash.

```
/* A / k → reciprocal_divide(A, R)
   precompute R = ((1LL << 32) + (k - 1)) / k */
u32 reciprocal_divide(u32 A, u32 R)
{
     return (u32)(((u64)A * R) >> 32);
}
```

Figure 11: Incorrect reciprocal multiplication optimization [21]. With this optimization `A/1` always produces zero, rather than `A`. The Linux kernel later disabled this optimization for BPF.

Figure 9 shows another example from BSD kernels: the interpreter needs to limit the jump offset k within the filter code, by checking if $pc + k$ exceeds the total length; otherwise an adversary can trick the kernel into executing illegal instructions. However, the interpreter misses the case where a large jump offset overflows $pc + k$ and bypasses the check.

3.2.2 Arithmetic errors

One infamous type of arithmetic errors is division by zero, which can crash the kernel if the interpreter fails to reject it. Figure 10 shows a bug where the interpreter tries to avoid that case, but performs the wrong check. Particularly, for BPF instructions A/k and A/X, one can observe their encoding difference by masking the opcode with `0x08`; the interpreter uses the wrong mask and fails to detect the case when k is zero.

```
/* BUG: k + sizeof(int32_t) can overflow */
if (k + sizeof(int32_t) > buflen)
    return 0;
A = EXTRACT_LONG(&p[k]);
```

Figure 12: Incorrect bounds check [51], which a large `k` can bypass since it overflows `k + sizeof(int32_t)` to a smaller value. A correct check is `k > buflen || sizeof(int32_t) > buflen - k`.

Optimizing arithmetic operations further complicates the situation. Figure 11 shows a bug in Linux's BPF JIT, which tries to optimize a division by a constant into a multiplication and a shift. This optimization, also used in the slab memory allocator, works well with input in that particular context (e.g., cache size as the divisor). However, this optimization is incorrect in general; for example, 65536/65537 should produce zero, but with the optimization the result becomes one. The Linux kernel has disabled this optimization for BPF [21].

3.2.3 Memory errors

An interpreter has access to two memory regions, the input packet P and the scratch memory M. It needs to correctly check the offsets of load and store instructions for both regions and reject illegal ones, which would otherwise trick the kernel into reading from or writing to memory that is out of range. The interpreter can easily miss such checks for some instructions [27], or more subtly, perform insufficient checks. Figure 12 shows an incorrect bounds check for `ld [k]`, where an adversary can use a large k to overflow and bypass the check, leading to illegal access beyond the input packet P.

3.2.4 Information leak

Since each BPF filter returns a 32-bit integer to user space, an interpreter needs to ensure that the return value is derived *only* from the input packet. In other words, it must *not* leak sensitive information from other processes nor the kernel. Several interpreters, including those in iOS (CVE-2012-3729) and Linux (CVE-2010-4158), allowed BPF filters to access uninitialized scratch memory M [30] or registers A and X [56], which could hold sensitive values from previous use. The interpreters fixed this vulnerability either by filling M, A, and X with zeros before execution, or by rejecting BPF filters that try to read these values before writing to them.

3.3 Kernel-user interface bugs

The logic at the kernel-user interface is straightforward: a user-space application encodes a BPF filter in bytecode format, as shown in Figure 4, and submits it to the kernel; the kernel decodes the bytecode and reconstructs the filter. Interestingly, there is still a possibility for programming mistakes such as copy-paste bugs [44]: for example, the Linux kernel once confused `BPF_W` with `BPF_B` for BPF bytecode encoding [4].

3.4 User-space bugs

It is tedious and error-prone to directly write BPF filters like Figure 2; for example, it requires programmers to correctly specify relative jump offsets. Many programmers instead express their policies through domain-specific tools or libraries, which provide a high-level interface for constructing filters. For example, invoking `tcpdump` with "`tcp dst port 80`" produces a 128-byte network filter for finding TCP packets sent to port 80. Applications like QEMU use the libseccomp library [22] to simplify the task of generating system call filters. These tools and libraries can submit incorrect filters to the kernel due to bugs in translating domain-specific policies into BPF filters [35], or when they try to optimize resulting filters [26].

3.5 Summary

Running user-specified code in the kernel offers flexibility and extensibility, at the price of a more vulnerable system. Achieving correctness and safety in an in-kernel interpreter is challenging: programmers can easily miss validating input for certain corner cases, or generate wrong code that is hard to notice. Many of these bugs have serious security impacts, as we have shown in Figure 7. In the next section, we will describe how to apply formal verification techniques to building Jitk, which is safe, fast, and immune to these bugs.

4 Overview

This section provides an overview of Jitk and its goals of correctly translating high-level, human-comprehensible policies to low-level native code for safe in-kernel execution. We describe Jitk in the context of the Seccomp system in Linux using the Jitk/BPF JIT, though the approach is general and not limited to BPF.

4.1 The architecture of Jitk/BPF

Figure 13 shows the architecture of Jitk/BPF. In contrast with the current Seccomp subsystem shown in Figure 1, there are three important differences.

System Call Policy Language (SCPL). Rather than manually writing BPF code, application developers can choose to specify system call policies using a high-level SCPL, which is more intuitive and helps programmers avoid mistakes in their policies. In steps 1 and 2, invoking the SCPL compiler on a SCPL program produces a corresponding BPF filter. As an example, Figure 3 shows the SCPL rules that capture the BPF filter used by OpenSSH shown in Figure 2, and our SCPL compiler will produce the latter from the former. Note that these two steps are optional; applications can still directly submit BPF bytecode to the kernel.

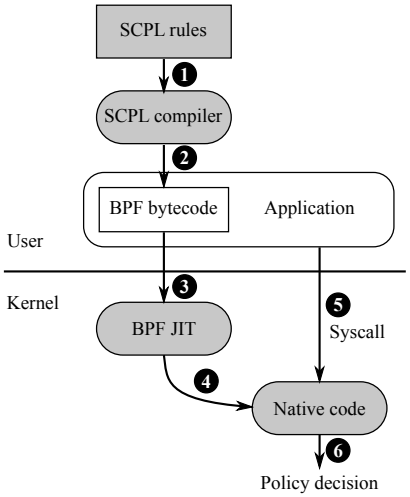

Figure 13: System overview of Jitk/BPF. Compared to the Seccomp subsystem in Linux shown in Figure 1, shaded components are introduced by Jitk. Steps are indicated with circled numbers.

JIT interpreter with a shared backend. In Jitk when the kernel accepts BPF bytecode from user space, a JIT translates the BPF filter into native code (steps 3 and 4). This native code is then executed for each system call to decide whether to allow that system call (steps 5 and 6). This JIT approach helps avoid the overhead of invoking the interpreter on each system call invocation. Jitk includes a compiler backend reused from CompCert, which is independent of BPF and can be shared among different interpreters. Our prototype implementation runs part of the JIT as a trusted user-space process (see §7).

Formal verification. The SCPL compiler and the BPF JIT are formally proven to be correct, as detailed next.

4.2 Goals

Jitk has two overall goals for enforcing user-specified policies in the kernel. First, well-behaved applications should be able to properly execute their filters in the kernel. That is, if an application developer writes down a set of SCPL rules, those rules should be correctly enforced by the kernel. We will call this the *correctness* goal. Second, it should be impossible for an adversary to misuse Jitk to "break" the kernel in any way. We will call this the *safety* goal. Jitk formalizes these goals in the form of a set of theorems and lemmas, as we will now discuss.

4.2.1 Correctness

The overall correctness goal required by an application that uses system call filtering is captured by the following end-to-end theorem:

Theorem 1 (End-to-end correctness)**.** For any system call policy p written in SCPL, if Jitk accepts it, the overall system enforces the semantics of p.

To enforce a system call policy in the kernel, Jitk needs to translate SCPL rules into BPF instructions, transmit BPF instructions from user space to the kernel, and translate BPF instructions into native code for execution in the kernel. To achieve Theorem 1, Jitk proves three lemmas that reflect this workflow.

First, the SCPL compiler must preserve the semantics of SCPL rules when generating BPF instructions:

Lemma 2 (SCPL-to-BPF semantic preservation)**.** Given a system call policy p written in SCPL, if the SCPL compiler translates it into a BPF filter f, f preserves the semantics of p:

$$\forall p : \mathit{SCPLc}(p) = \texttt{OK}\, f \implies p \approx f.$$

Here `OK` means the translation is successful; $\approx$ denotes semantic preservation.

Second, a filter must be transmitted correctly from user space to the kernel. To cross the user-kernel boundary, the filter is encoded from the in-memory representation into a byte-level representation as shown in Figure 4, submitted to the kernel through a system call (e.g., `prctl` in Linux [20]), and then decoded back into the in-memory representation by the kernel's BPF JIT. The reconstructed filter in the kernel must be the same as its user-space counterpart:

Lemma 3 (User-kernel representation equivalence)**.** If a BPF filter f is encoded into bytes in user space and the bytes are decoded back to a BPF filter in kernel space, f is preserved.

$$\forall f : \mathit{encode}(f) = \texttt{OK}\, b \implies \mathit{decode}(b) = \texttt{OK}\, f.$$

Finally, when the JIT translates BPF instructions into native code in the kernel, the native code must preserve the semantics of the BPF instructions:

Lemma 4 (BPF-to-native semantic preservation)**.** Given a BPF filter f, if the JIT accepts it and generates native code n, n preserves the semantics of f.

$$\forall f : \mathit{jit}(f) = \texttt{OK}\, n \implies f \approx n.$$

Jitk achieves this by first translating BPF to Cminor, an intermediate language in CompCert [42]. Jitk proves the correctness of the BPF-to-Cminor translation, and reuses Cminor-to-native from CompCert. See §5.1.2 for details.

Taken together, Lemmas 2 through 4 imply Theorem 1.

4.2.2 Safety

The safety concern is that an arbitrary user-space application should *not* be able to misuse Jitk to monopolize CPU time or to corrupt the kernel's memory. Particularly, both

native code generated by the BPF JIT and the JIT itself must be safe for in-kernel execution.

Although Theorem 1 (in particular Lemma 4) guarantees the correctness of native code with respect to given SCPL rules, it provides *no* guarantees that the generated native code will *not* cause an infinite loop or a stack overflow. The safety goal is captured by the following two theorems, which describe the temporal and spatial requirements of in-kernel execution, respectively.

Theorem 5 (Termination)**.** Given any BPF filter f, if the JIT accepts it and generates native code n, then n terminates.

$$\forall f : jit(f) = \texttt{OK}\, n \implies \texttt{terminate}(n).$$

Theorem 5 says if the JIT accepts an input BPF filter, the resulting native code must terminate, that is, the native code must come to a halt within a finite number of steps. We believe that termination is an appealing property for safety: it guarantees a bounded CPU time (i.e., no infinite loops), and *no* undefined behavior in the native code (e.g., no out-of-bounds memory access nor division by zero), with which the execution will get stuck.

Let S be a predefined parameter of the JIT, which indicates the maximum number of bytes that native code generated by the JIT can safely allocate and consume from the kernel stack.

Theorem 6 (Bounded stack usage)**.** Given any BPF filter f, if the JIT accepts it and generates native code n, n uses at most S bytes of stack.

$$\forall f : jit(f) = \texttt{OK}\, n \implies$$
any run of n uses at most S bytes of stack space.

Theorem 6 says that if the JIT accepts an input BPF filter, the resulting native code must *never* overflow the kernel stack.

The safety of the BPF JIT itself is guaranteed by Coq. The JIT is written in Coq (see §4.3); all Coq programs are guaranteed to provide memory safety, and are guaranteed to terminate, as it is impossible to write infinite loops in Coq's Gallina language [14: §7].

4.3 Development flow

To build a trustworthy in-kernel interpreter in Jitk, developers need to prove that an implementation satisfies the correctness and safety goals as formalized in §4.2. The development workflow is shown in Figure 14.

In particular, the JIT, the encoder-decoder from user space to kernel, and the SCPL compiler are all written in Coq. For each component, Jitk's Coq source code consists of three major parts: the specification, the implementation, and the proof that the implementation matches the specification. This source code is used in two ways. First, the Coq proof checker verifies that the proof is correct. Second, Coq transforms the implementation into OCaml. The generated OCaml code is compiled, together with a small I/O stub that performs I/O and invokes the generated code, into a native executable.

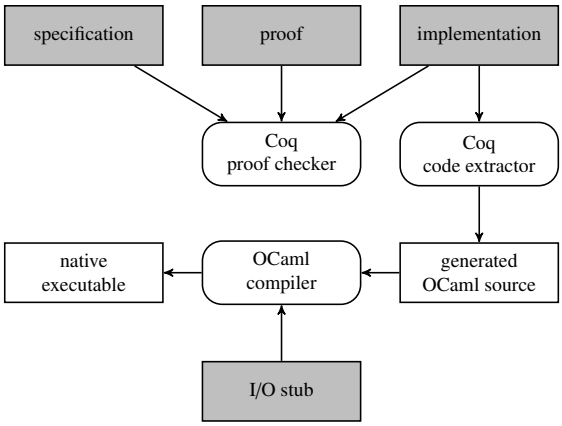

Figure 14: Development flow of Jitk using the Coq proof assistant. Shaded boxes indicate source code and proof written by developers.

5 Design

This section focuses on how the design and proofs help Jitk achieve its correctness and safety goals.

Figure 15 shows Jitk's detailed workflow and components, including the in-kernel JIT (§5.1), the high-level SCPL in user space (§5.2), the encoding-decoding across the two spaces (§5.3), and the integration of Jitk with Linux (§5.4). For each component, we will describe the specification, the implementation, and the proof.

5.1 JIT

A correct BPF JIT implementation should satisfy BPF-to-native semantic preservation (Lemma 4), termination (Theorem 5), and bounded stack usage (Theorem 6). To achieve these goals, we will start with the formal semantics of BPF (§5.1.1), and describe the design of the three key components: the translator (§5.1.2), which is responsible for transforming a BPF filter into native code, the checker (§5.1.3), which is responsible for making sure that all input filters are well-defined before being sent to the translator, and the validator (§5.1.4), which is responsible for ensuring bounded stack usage for output assembly code.

5.1.1 The specification

The specification of BPF consists of the syntax of instructions, the states, and the semantics, which is a set of state transitions among the states. The syntax mirrors the description in §3.1, which is omitted here. During execution, a BPF filter is in one of the following three states:

- initial state: a pair of current filter f and input packet P, denoted as $(Initialstate\ f\ P)$;

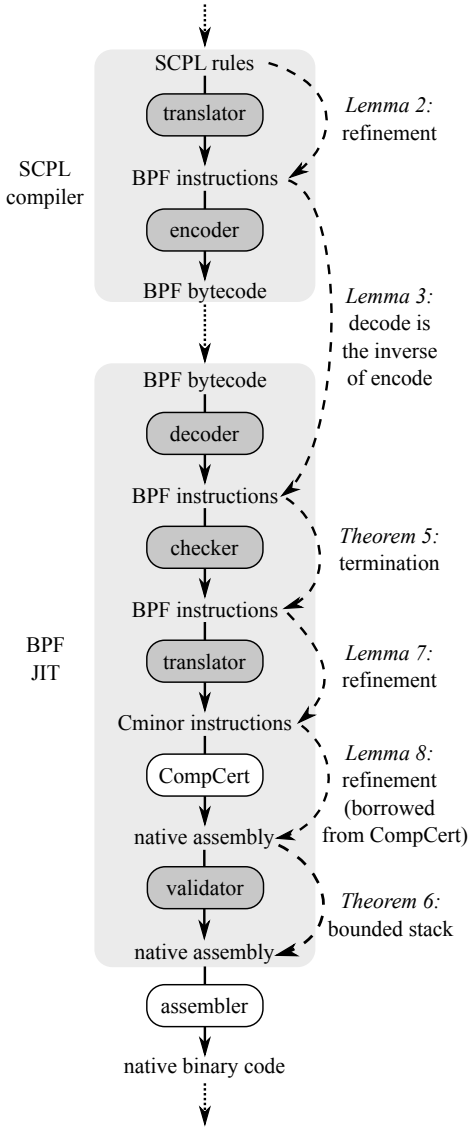

Figure 15: A detailed view of Jitk's SCPL compiler (in user space) and the BPF JIT (in kernel space).

- running state: a 6-tuple of registers A and X, scratch memory M, program counter pc, as well as f and P, denoted as (*State* A X M pc f P); and
- final state: a return value v, denoted as (*Finalstate* v).

A well-defined filter starts from the initial state, transitions through a set of running states, and halts in the final state. The first state transition is:

$$\begin{aligned}&(\mathit{Initialstate}\ f\ P)\\ \rightarrow\ &(\mathit{State}\ 0\ 0\ (\texttt{list\_repeat}\ N\ 0)\ f\ f\ P)\end{aligned} \tag{5.1}$$

This says that at start-up, A and X are set to zero, M is initialized as N zeros, and pc is set to the start of the filter f.

The core part of the semantics is transitions between two running states after executing an instruction other than `ret`. For example, the BPF instruction "`add` k" increases the value of register A by k, and the corresponding transition is:

$$\begin{aligned}&(\mathit{State}\ A\ X\ M\ (\texttt{add}\ k :: pc')\ f\ P)\\ \rightarrow\ &(\mathit{State}\ (\texttt{Int.add}\ A\ k)\ X\ M\ pc'\ f\ P)\end{aligned}$$

Here `add` $k :: pc'$ means that the current instruction to be executed is `add` k and the next program counter is pc'; `Int.add` is a 32-bit integer addition from CompCert. The specification says that after executing `add` k, the program transitions to a state with updated values of A and pc.

A more interesting example is a transition with a precondition. Below is the transition for BPF's unconditional jump instruction `ja` k:

$$\begin{aligned}&k < \texttt{length}\ pc' \implies\\ &(\mathit{State}\ A\ X\ M\ (\texttt{ja}\ k :: pc')\ f\ P)\\ \rightarrow\ &(\mathit{State}\ A\ X\ M\ (\texttt{skipn}\ k\ pc')\ f\ P)\end{aligned}$$

Here `length` returns the number of instructions remaining and `skipn` drops a given number of instructions. The specification says if k is less than the number of instructions remaining, then `ja` skips k instructions. Note that the specification says *nothing* about a too-large k with which `ja` could jump past the end of the filter. Therefore, such a filter is undefined, which a safe implementation like Jitk must reject.

The last state transition is to enter the final state with the return value k after executing `ret` k:

$$\begin{aligned}&(\mathit{State}\ A\ X\ M\ (\texttt{ret}\ \ k :: pc')\ f\ P)\\ \rightarrow\ &(\mathit{Finalstate}\ k)\end{aligned}$$

5.1.2 The translator and semantic preservation

The translator compiles a well-defined BPF filter (which passed the checker, as we will describe in §5.1.3) into native code. Our goal is to have an implementation with its correctness proof, as stated in Lemma 4. The key challenge of designing the translator is to choose an appropriate target language, which strikes a balance between the complexity of the implementation and the proof.

One approach is to directly produce low-level code such as x86 instructions from BPF, just like the existing JIT implementations in Linux and BSD kernels. The main downside of this approach, which we initially adopted, is that the big semantic gap between BPF and x86 makes it difficult to reason about the correspondence and prove properties about the target code. For example, consider translating BPF's unconditional jump `ja` k into x86's `jmp` n, where k and n are jump offsets in the corresponding languages. It is tricky to compute n (see §3.2.1); meanwhile, it is difficult to prove why n is the correct value that corresponds to k.

Jitk's translator targets Cminor, one of CompCert's intermediate languages [42]. Cminor can be considered

as an architecture-independent, simple C variant: it has only primitive types such as n-bit integers, and no pointer or struct types. The main advantage of targeting Cminor is that it retains high-level constructs, which simplifies the proof. For example, Cminor has labels and variables, with which one does not have to reason about low-level details such as jump offsets or the stack pointer.

To prove correctness of the translator, we prove that it satisfies the following property:

Lemma 7 (BPF-to-Cminor semantic preservation)**.** If the JIT translator generates Cminor code c from a BPF filter f, c preserves the semantics of f.

$$\forall f : jit_{translator}(f) = \texttt{OK}\, c \implies f \approx c.$$

The main technique for proving Lemma 7 is by simulation [43], as used throughout CompCert. Specifically, it suffices to show that each state transition in the BPF specification shown in §5.1.1 corresponds to some state transitions in the Cminor specification during translation. We omit the details.

A second advantage of targeting Cminor is that Jitk can reuse CompCert to transform Cminor code into native assembly. CompCert currently has support for x86, ARM, and PowerPC, for which it comes with its own semantic preservation theorem:

Lemma 8 (Cminor-to-native semantic preservation)**.** If CompCert generates native code n from a Cminor program c, n preserves the semantics of c.

$$\forall c : compcert(c) = \texttt{OK}\, n \implies c \approx n.$$

Together, Lemmas 7 and 8 imply our correctness goal, Lemma 4.

Finally, in order to transform the assembly code into native binary code, Jitk invokes a traditional assembler; CompCert does not include a provably correct assembler.

5.1.3 The checker and termination

Recall that Lemma 4 guarantees BPF-to-native semantic preservation: if an input filter terminates, the resulting native code produced by the translator also terminates. Therefore, to achieve the termination goal as stated in Theorem 5, it suffices to implement a checker that rejects all undefined input and ensures that any surviving filter terminates.

For BPF, this amounts to the following requirements: that all jump targets are forward (i.e., no loops), that all jump targets are in-bounds (i.e., not pointing past the end of the program), that all memory accesses are in-bounds (i.e., not reading or writing past the end of the input packet and the scratch memory), and that the input ends in a `ret` instruction. The combination of these rules ensures that every program that passes the checker will be well-defined, and will terminate with some return value according to BPF semantics.

We prove that the implementation of the checker satisfies the following property:

Lemma 9 (BPF termination)**.** If the JIT checker accepts a BPF filter f, then f terminates.

$$\forall f : jit_{checker}(f) = \texttt{OK}\, f \implies \texttt{terminate}(f).$$

Combining this with Lemma 4 gives our safety goal of termination, Theorem 5. Note that it is impossible to miss any undefined cases in the implementation of the checker, otherwise the proof of Lemma 9 would not succeed.

5.1.4 The validator and bounded stack usage

CompCert does not provide facilities for reasoning about stack bounds across transformations. In order to prove Theorem 6, one option is to extend CompCert with proposed support for tracking stack bounds [8], which would allow Jitk to prove a theorem about it.

Jitk adopts a simpler approach using the validator. As for BPF, there is only one function with a fixed number of variables and a fixed-size object (scratch memory) on the stack. The validator inspects the size in the stack frame allocation instruction at function entry in the resulting assembly code, and fails the JIT if the size is larger than a predefined S. It is then easy to prove Theorem 6, as any generated assembly code that passes the validator uses at most S bytes from the stack space.

5.2 SCPL

The design of our SCPL is inspired by the libseccomp API [22]; the key difference is that the SCPL compiler is provably correct. As shown in Figure 3, developers specify rules for matching different system calls (and optionally system call arguments), along with actions to take when those rules match (e.g., allow the system call, or reject it with a particular `errno` value). There is also a default action, if none of the rules match. The formal specification of SCPL is similar to BPF described in §5.1.1: the syntax, the states, and the state transitions. The proof of SCPL-to-BPF correctness is also similar to BPF-to-Cminor. We omit the details here.

5.3 Encoding and decoding

To ensure that BPF programs are faithfully encoded and decoded when transmitted across the user-kernel space boundary, Jitk implements an encoder and decoder, which transforms an in-memory representation of a BPF program into BPF bytecode, and back into an in-memory representation.

One option for proving the correctness of the BPF encoder and decoder would be to define semantics for BPF bytecode, as defined by byte-level sequences, and to prove equivalence between the in-memory representation (under

the semantics of in-memory BPF programs) and the byte-level representation (under the semantics of byte-level encodings). However, this approach is quite cumbersome, owing to the complexity of defining the semantics of BPF programs at the low level of byte encodings.

Instead, Jitk takes a pragmatic alternative: it proves that the decoder is the inverse of the encoder (Lemma 3). When used in combination with SCPL, this guarantees that SCPL-generated BPF bytecode will necessarily be decoded correctly by the Jitk/BPF JIT in the kernel. It does not guarantee that the encoder and decoder implemented by Jitk are compatible with any other encoding (e.g., the encoding produced by libseccomp). In practice, we believe this is not a significant problem, because the specification of the correct encoding and decoding would be of similar complexity to our current Coq encoder/decoder implementation, and our theorem (Lemma 3) provides a strong sanity check on the internal consistency of our encoder and decoder.

5.4 Linux kernel integration

The Linux kernel interacts with Jitk in two ways. The first is when an application submits a BPF filter to Seccomp through the `prctl` system call [20]: we modified the kernel to invoke Jitk's BPF JIT. The JIT translates the filter into native code; if the translation fails, indicating that the BPF filter code was not well-formed, the `prctl` system call returns an error.

The second is when an application makes a subsequent system call: we modified the kernel to check if there is already a JIT-compiled filter associated with the process; if so, the kernel treats the filter as a function pointer, invoking it with a single argument (the structure containing the system call number and arguments, shown in Figure 6).[1] The return value determines the resulting action for this system call, much as with the existing Seccomp design.

6 Discussion

There are some mistakes that Jitk's theorems cannot prevent. First, if the specifications of BPF and SCPL are buggy, then Jitk's JIT implementations can have bugs.

Another example is an overly strict checker, such as NetBSD #37663 [28], which rejected any BPF program that used the multiply instruction. Such an implementation does not violate either correctness or safety goals, since all of our theorems are conditional on our system accepting a given program. It would be possible to prove an additional theorem that required Jitk/BPF to accept certain programs; one good candidate would be a requirement that Jitk/BPF accept all BPF programs generated by the SCPL compiler.

[1] Linux kernel 3.16 or later does not require this modification anymore, as it has been changed to work in the same way.

Jitk's theorems also cannot prevent the use of Jitk's JIT for JIT spraying [47], which can make it easier to exploit memory corruption vulnerabilities in the rest of the Linux kernel. Given a formal set of requirements for a JIT to mitigate the effects of JIT spraying (e.g., ensuring that a constant in the input bytecode does not appear in the output code), it may be possible to prove that Jitk correctly implements such mitigation techniques.

Jitk's encoder/decoder can have self-consistent mistakes, in that the encoder and decoder are consistent with each other (satisfying Lemma 3), but do not match the encoding used by others for the same bytecode. We believe it is unlikely for the reasons discussed in §5.3.

Finally, Jitk assumes several additional parts are correct without a formal proof. First, the Coq proof checker is assumed to be correct. While bugs have been found in Coq, we believe Coq provides a strong degree of assurance that Jitk is trustworthy. The Coq extraction system and the OCaml compiler and runtime are also assumed to be correct. We believe this is reasonable because Coq itself is written in OCaml. That said, bugs in Coq's extraction system have been found in the past [57, 59].

Jitk's OCaml I/O stub has no proof of correctness. It is about 70 lines of code, and performs simple operations: taking input from stdin, passing it into the Coq-extracted code, and printing the results to stdout.

The rest of the kernel code, including the wrapper that invokes the Jitk JIT and that invokes the filter code produced by the JIT, is assumed to be correct. Particularly, Jitk assumes that the kernel does not trample on the JIT itself, that the kernel correctly interprets the results from the JIT and the filter, and that the kernel synthesizes an correct input packet to the filter, namely, a single pointer argument pointing to a valid memory region whose size matches the structure shown in Figure 6.

Jitk also assumes that the kernel invokes the JIT-compiled code with an appropriate calling convention. For example, on x86 the JIT-compiled code assumes that the input argument is passed on the stack, as CompCert's x86 backend uses the *cdecl* calling convention; however, the Linux kernel uses *fastcall* by default (e.g., with gcc's `-mregparm=3`), which passes the input argument in the `EAX` register. We bridged the gap using a function wrapper.

Finally, Jitk assumes that CompCert generates correct assembly code for the filter, which is a single-argument function. One technical complication is that CompCert's semantics are defined only for complete programs that take *no* arguments. This precludes even well-formed C programs with a `main` function that takes two arguments, `argc` and `argv`, for which theoretically CompCert provides no guarantees. We believe this is not a likely source of bugs in practice.

Component	Lines of code
Specifications (SCPL, BPF)	420 lines of Coq
Implementation (SCPL, BPF)	520 lines of Coq
Proof (SCPL, BPF)	2,300 lines of Coq
Extraction to OCaml	50 lines of Coq
I/O stub	70 lines of OCaml
Linux kernel changes	150 lines of C
Total	3,510 lines of code

Figure 16: Lines of code for our Jitk/BPF prototype.

7 Implementation

We have implemented a fully functional prototype of Jitk for Linux's BPF-based Seccomp system. The breakdown of our Jitk/BPF prototype (excluding the components borrowed from CompCert) in terms of lines of code is shown in Figure 16.

As mentioned in §4.3, the BPF JIT is written in Coq. We extracted the Coq implementation into OCaml code, and linked it into an executable, together with the I/O stub and the OCaml runtime. In order to run this executable for translating BPF bytecode into native binary code, we chose to put it in user space and modified the kernel to perform an upcall, using Linux's `call_usermodehelper` interface. Putting the executable with the OCaml runtime into the kernel would be doable, as demonstrated by the Mirage unikernel [45], but would unnecessarily complicate our implementation. Trusting a user-mode process running as root to produce native binary code for running in the kernel seems reasonable, given that root processes can also load arbitrary kernel modules.

Jitk supports generating x86, ARM, and PowerPC code, as CompCert provides the corresponding backends. Jitk does not support architectures that CompCert lacks, such as x86-64.

Like CompCert, Jitk relies on a conventional (not proven correct) assembler to convert textual assembly code into a native binary. Jitk uses the GNU assembler `as` for this purpose, and disables assembler-level optimizations (using the `-n` option) out of precaution.

8 Evaluation

In our evaluation, we aim to answer five questions:

- How much effort does it take to build Jitk? (§8.1)
- Does formal verification prevent the kinds of bugs that arise in practice? (§8.2)
- Does Jitk's JIT produce efficient filter code? (§8.3)
- How much effort is required to use SCPL? (§8.4)
- What is the end-to-end performance of Jitk? (§8.5)

8.1 Verification effort

The total effort to build Jitk for Seccomp was shown in Figure 16. Much of the effort went into constructing proofs. The 2,300 lines of Coq proof code consist of 650 lines of general helpers (the `crush` tactic from CPDT [14] and miscellaneous lemmas about linked lists and arithmetic), 950 lines of refinement proof (that the BPF JIT preserves semantics), 350 lines of termination proof (that programs that pass the checker are well-defined), 150 lines of encoding proof (that decoding is the inverse of encoding), and 250 lines of proof for the SCPL compiler.

To determine if this approach can be applied to another bytecode language, we implemented a JIT for the INET-DIAG interpreter from the Linux kernel. INET-DIAG has a simpler bytecode language, and the code and proof sizes were correspondingly smaller, totaling 1,590 lines of code. Overall, we believe this indicates Jitk is a practical approach for building trustworthy in-kernel interpreters.

8.2 Bug case study

To evaluate whether Jitk's formal verification does a good job of preventing bugs that arise in practice, we perform an analysis of the bugs that have been found in interpreters so far (Figure 7), and manually determine whether such a bug could have been present in an implementation that provably satisfies Jitk's theorems. Our results show that Jitk is effective at preventing these bugs.

Control-flow errors. The control flow errors described in §3.2.1, such as misaligned jump targets and overflow in computing the jump offset, cannot occur in Jitk, since Lemma 4 guarantees that all jumps in native code preserve the semantics. It is also impossible to "run off the end" of generated native code, since Theorem 5 guarantees that every BPF program that passes the checker must terminate. We found and fixed several off-by-one jump errors in our implementation while proving Lemma 7.

Arithmetic errors. If the JIT forgets to check for division by zero, the proof of Lemma 7 cannot succeed: in Cminor, division by zero is undefined, and it would be impossible to prove that the generated Cminor code refines the semantics of the BPF program. Similarly, if the JIT has an incorrect optimization, such as `reciprocal_divide`, the proof of Lemma 7 cannot proceed, either. We initially forgot to check for division by zero, and had to address it in order to complete the proof.

Memory errors. If the memory index in a BPF instruction is in-bounds, the generated Cminor instruction must access the same memory index; otherwise the proof of Lemma 7 cannot proceed. This eliminates incorrect memory indices. On the other hand, if the memory index in a BPF instruction is out-of-bounds, the BPF program is undefined, and the termination proof ensures that the checker

	BPF	x86		ARM		PowerPC	
		FreeBSD	Jitk	Linux	Jitk	Linux	Jitk
OpenSSH	312	466	274 (58.8%)	384	396 (103.1%)	452	340 (75.2%)
vsftpd	576	1,177	443 (37.6%)	616	624 (101.3%)	736	548 (74.5%)
Native Client	664	928	626 (67.5%)	828	844 (101.9%)	984	688 (69.9%)
QEMU	1,680	2,364	2,037 (86.2%)	2,048	2,156 (105.3%)	2,780	1,840 (66.2%)
Firefox	1,720	2,314	1,636 (70.7%)	2,164	2,196 (101.5%)	2,564	1,748 (68.2%)
Chrome	2,144	4,640	2,122 (45.7%)	2,380	2,348 (98.7%)	3,260	2,308 (70.8%)
Tor	3,032	6,841	2,691 (39.3%)	3,400	4,048 (119.1%)	4,308	3,304 (76.7%)

Figure 17: Comparison of sizes of native code generated by the BPF JIT of Jitk, the Linux kernel, and the FreeBSD kernel, measured in bytes. "BPF" lists the size of BPF filters, also in bytes.

must reject all such programs. Thus, out-of-bounds memory accesses are avoided.

Information leak. The initial state transition (5.1) specifies that the initial state of a BPF program contain zeroes in all registers and scratch memory locations. The proof of Lemma 4 guarantees that no generated native code can produce results inconsistent with zeroed initial memory. Note that the lemma leaves open the option of not initializing these memory locations, as long as they are not actually read by the BPF program. In fact, the CompCert compiler's optimization passes will eliminate initialization of unused memory locations in a provably correct way.

Encoding and decoding bugs. Lemma 3 guarantees that the decoder is the inverse of the encoder. This lemma's proof cannot go through if one of the encoder or decoder has a bug, as was the case in a recent Linux issue [4].

Bugs in BPF generation tools. If developers write SCPL rules and invoke the SCPL compiler to generate BPF filters, Lemma 2 guarantees the absence of incorrect filters, unlike with tcpdump or libseccomp.

8.3 Code quality

To understand the quality of native code generated by Jitk, we collect BPF filters used by popular applications shipped in Ubuntu 14.04. To extract these filters, we use `LD_PRELOAD` to intercept the `prctl` system call, used to submit system call filters to the kernel. We then compare the sizes of resulting native code produced by Jitk/BPF and by two widely used in-kernel BPF JITs, Linux and FreeBSD, as shown in Figure 17 (though Linux does not use the BPF JIT for Seccomp).

For ARM and PowerPC, we compare Jitk with Linux's BPF JIT (FreeBSD's does not support the two architectures). For x86, we compare Jitk with FreeBSD's BPF JIT (Linux's does not support x86). Also, the current Linux JIT (both ARM and PowerPC) failed on one special BPF instruction used by Seccomp; we patched it for this comparison. The results show that in addition to the correctness and safety guarantees, the quality of the native code produced by Jitk is comparable to that from existing in-kernel JITs. Particularly, Jitk generates substantially smaller code than existing in-kernel JITs on x86 and PowerPC. After inspecting the resulting assembly code, we believe the reason is that Jitk is built on top of CompCert, which performs more comprehensive and effective optimizations (e.g., common-code elimination).

8.4 SCPL

To evaluate the usability of SCPL, we translated the Seccomp policy used by OpenSSH from raw BPF operations specified by the developer into an SCPL policy. The resulting SCPL policy was 40 lines of code, parts of which were shown in Figure 3. Integrating the SCPL policy into OpenSSH required changing an additional 20 lines of OpenSSH source code, to load the BPF filter produced by the SCPL compiler, instead of using the manually written BPF filter. Overall, we believe this suggests that SCPL is easy to use in real applications.

8.5 Performance

To evaluate the performance of Seccomp with Jitk, we measured the performance of OpenSSH running on an i386 Linux system. We measured two OpenSSH configurations: one with the hand-written BPF filter, and one with the SCPL-generated filter (as described above). We also considered two kernel configurations: one using the stock Linux kernel, with its unverified BPF interpreter, and one using our modified Linux kernel that uses Jitk's BPF JIT. Since the Seccomp policy in OpenSSH applies to the process that performs user authentication, we measured the time it takes to log in via SSH and then immediately disconnect, from the same machine (i.e., measuring the time for `ssh localhost exit`), using RSA key authentication.

Figure 18 shows the results on a single-core 2.8 GHz Intel Xeon CPU with 3 GB DRAM running a 32-bit Linux 3.15-rc1 kernel. As can be seen, SCPL-generated filters perform just as well as the hand-written BPF filter in OpenSSH. Jitk's BPF JIT introduces about 20 msec of additional latency; this is due to the overhead of invoking a new process for the OCaml runtime and the assembler. We measured the time taken just to install the OpenSSH

	Stock OpenSSH BPF filter	SCPL-generated BPF filter
Stock Linux	124 msec	124 msec
Jitk BPF JIT	144 msec	144 msec

Figure 18: Time taken to login and disconnect from an OpenSSH server in different configurations; using SCPL gives the same performance as hand-written BPF filters.

BPF filter as a Seccomp policy in Linux, using the `prctl` system call; the time with the stock Linux kernel was 1 msec, and the time with Jitk's BPF JIT was 21 msec; the time to just run the BPF JIT's OCaml binary on that BPF filter is 14 msec. We believe this can be reduced further by using a persistent user-space helper process instead of spawning a new process for every BPF filter.

One benefit of Jitk's BPF JIT over a traditional interpreter is that once the BPF filter has been translated into native code, subsequent system calls can execute with lower overhead. To measure this, we used the BPF filter from OpenSSH, and measured the time to both install the BPF filter, and to perform 1,000,000 `gettimeofday` system calls (we moved `gettimeofday` to be the last system call allowed by the BPF filter). With the stock Linux BPF interpreter, this took 771 msec; with Jitk's BPF JIT, this took 691 msec; without any filter, this took 460 msec.

9 Conclusion

Jitk is a new approach for building in-kernel JIT interpreters that guarantee functional correctness using formal verification techniques. Jitk guarantees correctness through high-level policy rules in user-space applications, to lower-level BPF, across the user-kernel space boundary, and to native code in-kernel. It also guarantees termination and bounded stack usage for native code executed in-kernel. An analysis of known interpreter vulnerabilities demonstrates that Jitk prevents all classes of security vulnerabilities discovered in existing kernel interpreters. An experimental evaluation shows that Jitk's SCPL rules are easy to integrate into existing applications, and that Jitk achieves good end-to-end performance. We believe that this is a promising direction since it achieves flexibility, safety, and good performance. All of Jitk's source code is publicly available at http://css.csail.mit.edu/jitk/.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Gernot Heiser, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[2] Bitcoin. Script, 2014. https://en.bitcoin.it/wiki/Script.

[3] D. Borkmann. net: sched: cls_bpf: add BPF-based classifier, Oct. 2013. http://patchwork.ozlabs.org/patch/286589/.

[4] D. Borkmann. net: filter: seccomp: fix wrong decoding of `BPF_S_ANC_SECCOMP_LD_W`, Apr. 2014. http://patchwork.ozlabs.org/patch/339039/.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, Oct.–Nov. 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, June–July 2004.

[8] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, Edinburgh, UK, June 2014.

[9] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, Big Sky, MT, Oct. 2009.

[10] H. Chen, C. Cutler, T. Kim, Y. Mao, X. Wang, N. Zeldovich, and M. F. Kaashoek. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, Singapore, July 2013.

[11] D. Chisnall. LLVM in the FreeBSD toolchain. In *Proceedings of AsiaBSDCon*, pages 13–20, Tokyo,

Japan, Mar. 2014.

[12] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[13] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402, Boston, MA, Sept. 2013.

[14] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, Nov. 2013.

[15] J. Corbet. A JIT for packet filters, Apr. 2011. https://lwn.net/Articles/437981/.

[16] J. Corbet. BPF tracing filters, Dec. 2013. https://lwn.net/Articles/575531/.

[17] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 351–366, Stevenson, WA, Oct. 2007.

[18] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 59–72, San Diego, CA, Dec. 2008.

[19] der Mouse. NetBSD PR #3366: bpf doesn't check its filter enough, Mar. 1997. http://gnats.netbsd.org/3366.

[20] W. Drewry. SECure COMPuting with filters, Jan. 2012. http://lwn.net/Articles/498231/.

[21] E. Dumazet. bpf: do not use reciprocal divide, Jan. 2014. http://patchwork.ozlabs.org/patch/311163/.

[22] J. Edge. A library for seccomp filters, Apr. 2012. http://lwn.net/Articles/494252/.

[23] K. Elphinstone and G. Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–150, Farmington, PA, Nov. 2013.

[24] D. R. Engler, M. F. Kaashoek, and J. 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)*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Seattle, WA, Nov. 2006.

[26] G. Harris. Issue #38: Another optimization bug, Dec. 2003. https://github.com/the-tcpdump-group/libpcap/issues/38.

[27] G. Harris. NetBSD PR #32198: `bpf_validate()` needs to do more checks, Nov. 2005. http://gnats.netbsd.org/32198.

[28] G. Harris. NetBSD PR #37663: `bpf_validate` rejects valid programs that use the multiply instruction, Jan. 2008. http://gnats.netbsd.org/37663.

[29] G. Harris. NetBSD PR #43185: `bpf_validate()` uses `BPF_RVAL()` when it should use `BPF_SRC()`, Apr. 2010. http://gnats.netbsd.org/43185.

[30] G. Harris. NetBSD PR #45412: `bpf_filter()` can leak kernel stack contents, July 2011. http://gnats.netbsd.org/45412.

[31] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE Dependable Systems and Networks Conference*, pages 33–42, Lisbon, Portugal, June–July 2009.

[32] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification, Dec. 2011. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf.

[33] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, Oct. 2005.

[34] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, Saint-Malo, France, Oct. 1997.

[35] B. Keats. TCPDUMP 3.9.4 under Fedora Core 5 seems to generate the wrong BPF for `DLT_PRISM_HEADER`, Aug. 2006. http://seclists.org/tcpdump/2006/q3/37.

[36] M. Kerrisk. LCA: The Trinity fuzz tester, Feb. 2013. https://lwn.net/Articles/536173/.

[37] J.-u. Kim. Add experimental BPF Just-In-Time compiler for amd64 and i386, Dec. 2005. http://docs.freebsd.org/cgi/mid.cgi?200512060258.jB62wCnk084452.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[39] M. Koetter. Linux 3.0 bpf jit x86_64 exploit, Dec. 2011. http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit.

[40] A. Kuznetosv. SS utility: Quick intro, Sept. 2001. http://www.cyberciti.biz/files/ss.html.

[41] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of 16th International Symposium on Formal Methods*, pages 806–809, Eindhoven, the Netherlands, Nov. 2009.

[42] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[43] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, Dec. 2009.

[44] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Francisco, CA, Dec. 2004.

[45] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, Houston, TX, Mar. 2013.

[46] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 115–128, Cascais, Portugal, Oct. 2011.

[47] K. McAllister. Attacking hardened Linux systems with kernel JIT spraying, Nov. 2012. http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[48] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 259–270, San Diego, CA, Jan. 1993.

[49] M. McCoyd, R. Krug, D. Goel, M. Dahlin, and W. Young. Building a hypervisor on a formally verified protection layer. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 5069–5078, Maui, HI, Jan. 2013.

[50] M. O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, pages 107–118, Madrid, Spain, Jan. 2011.

[51] A. Nasonov. NetBSD PR #45751: No overflow check in `BPF_LD|BPF_ABS`, Dec. 2011. http://gnats.netbsd.org/45751.

[52] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, WA, Oct. 1996.

[53] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–227, Seattle, WA, Oct. 1996.

[54] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

[55] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, Dec. 1999.

[56] A. Starovoitov. net: filter: initialize A and X registers, Apr. 2014. http://patchwork.ozlabs.org/patch/341693/.

[57] F. Tuong. Bug 2570 - in extraction optimization, a eta-reduction leads to a not generalizable '_a, July 2011. https://coq.inria.fr/bugs/show_bug.cgi?id=2570.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.

[59] R. Zumkeller. Bug 843 - extraction breaks module typing, Aug. 2004. https://coq.inria.fr/bugs/show_bug.cgi?id=843.