Jitk: A trustworthy in-kernel interpreter infrastructure

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, Zachary Tatlock
MIT and University of Washington
Modern OSes run untrusted user code in kernel

- In-kernel interpreters
  - Seccomp: sandboxing (Linux)
  - BPF: packet filtering
  - INET_DIAG: socket monitoring
  - Dtrace: instrumentation
- (RAR, Bitcoin, ClamAV, Python re, ...)

- Critical to overall system security
  - Any interpreter bugs are serious!
Many bugs have been found in interpreters

- Kernel space bugs
  - Control flow errors: incorrect jump offset, ...
  - Arithmetic errors: incorrect result, ...
  - Memory errors: buffer overflow, ...
  - Information leak: uninitialized read
- Kernel-user interface bugs
  - Incorrect encoding/decoding
- User space bugs
  - Incorrect input generated by tools/libraries
- Some have security consequences: CVE-2014-2889, ...

See our paper for a case study of bugs
How to get rid of all these bugs at once?
Theorem proving can help kill all these bugs

- seL4: provably correct microkernel [SOSP'09]
- CompCert: provably correct C compiler [CACM'09]
- This talk: Jitk
  - Provably correct interpreter for running untrusted user code
  - Drop-in replacement for Linux's seccomp
  - Built using Coq proof assistant + CompCert
Theorem proving: overview

- Proof is machine-checkable: Coq proof assistant
- Proof: correct specification $\Rightarrow$ correct implementation
- Specification should be much simpler than implementation
Challenges

- What is the specification?
- How to translate systems properties into proofs?
- How to extract a running system?
Contributions & outline

- Specifications: capture systems properties
- Theorems: ensure correctness of implementation
- Integrate Jitk with Linux kernel
Seccomp: reduce allowed syscalls

- 1: app submits a Berkeley Packet Filter (BPF) to kernel at start-up
  - Example: if syscall is open, return some errno
  - App cannot open new files, even if it's compromised later
- 2: kernel BPF interpreter executes the filter against every syscall
- 3: kernel decides whether to allow/deny the syscall based on result

Diagram:

1. User submit BPF bytecode to kernel
2. Kernel runs BPF bytecode on every syscall
3. Kernel makes policy decision based on syscall result
Seccomp/BPF example: OpenSSH

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

- Deny open() with errno EACCES
- Allow gettimeofday(), ...
- Kill the current process if seeing other syscalls
Summary of seccomp

- Security critical: sandboxing mechanism
- Widely used: by Chrome, OpenSSH, QEMU, Tor, ...
- Performance critical: invoked for each syscall
- Non-trivial to do right: many bugs have been found
- General: similar design found in multiple OS kernels
Specification: what seccomp should do

Goal: enforce user-specified syscall policies in kernel

- What kernel executes is what user specifies
  - Kernel: BPF-to-x86 for execution
  - BPF transferred from user space to kernel
  - User space: write down policies as BPF

- Non-interference with kernel
  - Termination: no crash nor infinite loop
  - Bounded stack usage: no kernel stack overflow
Jitk 1/3: BPF-to-x86 for execution

JIT: translate BPF to x86 for in-kernel execution

- JIT is error-prone: CVE-2014-2889

```c
jcc = ...; /* conditional jump opcode */
if (filter[i].jf)
    true_offset += is_near(false_offset) ? 2 : 6;
EMIT_COND_JMP(jcc, true_offset);
if (filter[i].jf)
    EMIT_JMP(false_offset);
```

- Goal: Jitk's output x86 code preserves the behavior of input BPF
- x86 code cannot have buffer overflow, control-flow bugs, ...
BPF-to-x86 correctness: state machine simulation

- Model BPF and x86 as two state machines: by reading manuals
  - BPF state: 2 regs, fixed-size memory, input, program counter
  - BPF instruction: state transition
  - x86: [...] - reused from CompCert

- Theorem (backward simulation):
  If JIT succeeds, every state transition in output x86 corresponds to some state transition(s) in input BPF.
Jitk's approach for BPF-to-x86

- Strawman: write & prove BPF-to-x86 translator
  - Backward simulation is hard to prove
  - Big semantic gap between BPF and x86

- Prove forward simulation and convert
  - Every state transition in BPF corresponds to some state transition(s) in output x86
  - Conversion possible if lower level (x86) is deterministic

- Add intermediate languages between BPF and x86
  - Choose Cminor ("simpler" C) from CompCert as detour
  - BPF-to-x86: BPF-to-Cminor + CompCert's Cminor-to-x86
Jitk 2/3: user-kernel interface correctness

- App submits BPF in bytecode from user space to kernel
- Kernel decodes bytecode back to BPF - bugs happened!

Goal: BPF is correctly decoded in kernel

- Alternative approach: state machine simulation
  - Spec: state machine for bytecode representation
  - Simulation: bytecode BPF ↔ BPF
  - Challenge: spec is as complex as implementation
Jitk's approach: user-kernel BPF equivalence

- Two functions: `encode()` and `decode()`
- Choose a much simpler spec: equivalence

\[
\forall f: \text{encode}(f) = b \Rightarrow \text{decode}(b) = f
\]

- Trade-off: can have "consistent" bugs
  - `encode()` and `decode()` could make the same mistake
  - `decode()` could behave differently from existing BPF
Jitk 3/3: input BPF correctness

Goal: input BPF is "correct"

<table>
<thead>
<tr>
<th>BPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [0]</td>
</tr>
<tr>
<td>jeq #SYS_open, L1, L2</td>
</tr>
</tbody>
</table>

L1: ret #RET_ERRNO#EACCES
L2: jeq #SYS_gettimeofday, L3, L4
L3: ret #RET_ALLOW
L4: ...

ret #RET_KILL

; load syscall number
; deny open() with errno = EACCES
; allow gettimeofday()
; default: kill current process

- Does this BPF correctly implement policies?
- Is the BPF spec correct?
Jitk's approach: add a higher level

SCPL: domain-specific language for writing syscall policies

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

- Much simpler than BPF → unlikely to make mistakes
- SCPL-to-x86 = SCPL-to-BPF + BPF-to-x86
  - Proof: state machine simulation
  - Use SCPL: don't need to trust BPF spec
  - Improve confidence in BPF spec
Summary of Jitk's approaches

- State machine simulation: BPF-to-x86 and SCPL-to-BPF
  - Add extra levels in-between to bridge gap
  - Forward simulation to backward simulation
  - More abstraction, more confidence
- Equivalence: user-kernel data passing
  - Trade-off: simpler spec vs. can have "consistent" bugs
Development: write shaded boxes

specification -> proof -> Coq proof checker -> OCaml compiler -> generated OCaml source

implementation -> Coq code extractor -> native executable

I/O stub
Integrate Jitk (shaded boxes) with Linux kernel

- Modify Linux kernel to invoke BPF-to-x86 translator
  - Run the translator as a trusted user-space process
  - The translator includes OCaml runtime & GNU assembler
- Modify Linux kernel to invoke output x86 code for each syscall
Jitk's theorems can stop a large class of bugs

Manually inspected existing bugs

- Kernel space bugs: BPF-to-x86 correctness
  - ✔ Control flow errors
  - ✔ Arithmetic errors
  - ✔ Memory errors
  - ✔ Information leak
- Kernel-user interface bugs: user-kernel BPF equivalence
  - ✔ Incorrect encoding/decoding
- User space bugs: SCPL-to-BPF correctness
  - ✔ Incorrect input generated by tools/libraries
What Jitk's theorems cannot stop

- Over-strict: Jitk could reject correct input SCPL/BPF
- Side channel: JIT spraying attacks
- Bugs in specifications: SCPL, BPF, x86
- Bugs in CompCert's TCB: Coq, OCaml runtime, GNU assembler
- Bugs in other parts of Linux kernel
Evaluation

- How much effort does it take to build Jitk?
- What is the end-to-end performance?
- Does Jitk’s JIT produce efficient x86 code?
Building effort is moderate

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications (SCPL, BPF)</td>
<td>420 lines of Coq</td>
</tr>
<tr>
<td>Implementation (SCPL, BPF)</td>
<td>520 lines of Coq</td>
</tr>
<tr>
<td>Proof (SCPL, BPF)</td>
<td>2,300 lines of Coq</td>
</tr>
<tr>
<td>Extraction to OCaml</td>
<td>50 lines of Coq</td>
</tr>
<tr>
<td>I/O stub</td>
<td>70 lines of OCaml</td>
</tr>
<tr>
<td>Linux kernel changes</td>
<td>150 lines of C</td>
</tr>
<tr>
<td>Total</td>
<td>3,510 lines of code</td>
</tr>
</tbody>
</table>
End-to-end performance overhead is low

- OpenSSH on Linux/x86
  - Stock Linux: interpreter (no x86 JIT support)
  - Jitk: JIT
- Jitk's BPF-to-x86 one-time overhead: 20 msec per session
- Time for 1M gettimeofday syscalls: smaller is better (in msec)
Jitk produces good (often better) code

Output x86 code size comparison (smaller is better)

- Existing BPF JITs have very limited optimizations
- Jitk leverages optimizations from CompCert
Related work

- Theorem proving: seL4, CompCert
- Model checking & testing: EXE, KLEE
- Microkernel, SFI, type-safe languages
Conclusion

Jitk: run untrusted user code in kernel with theorem proving

- Strong correctness guarantee
- Good performance
- Approaches for proving systems properties