Hyperkernel: Push-Button Verification of an OS Kernel

By Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, Xi Wang

Presented by Young Li

Formal Verification

- Write a *specification* of expected behavior
- Prove that the implementation matches the specification

Proof Burden

Proof burden of verifying OS kernels is high

- CertiKOS: 1-2 person-years (2016)
- seL4: 11 person-years (2009)

Can we make this cost lower so that verification is scalable?

Contributions

1. Push-button verification workflow for OS kernels
2. Kernel interface design amenable to SMT solving

Push-button verification workflow for OS kernels

Toolchain: CertiKOS and seL4

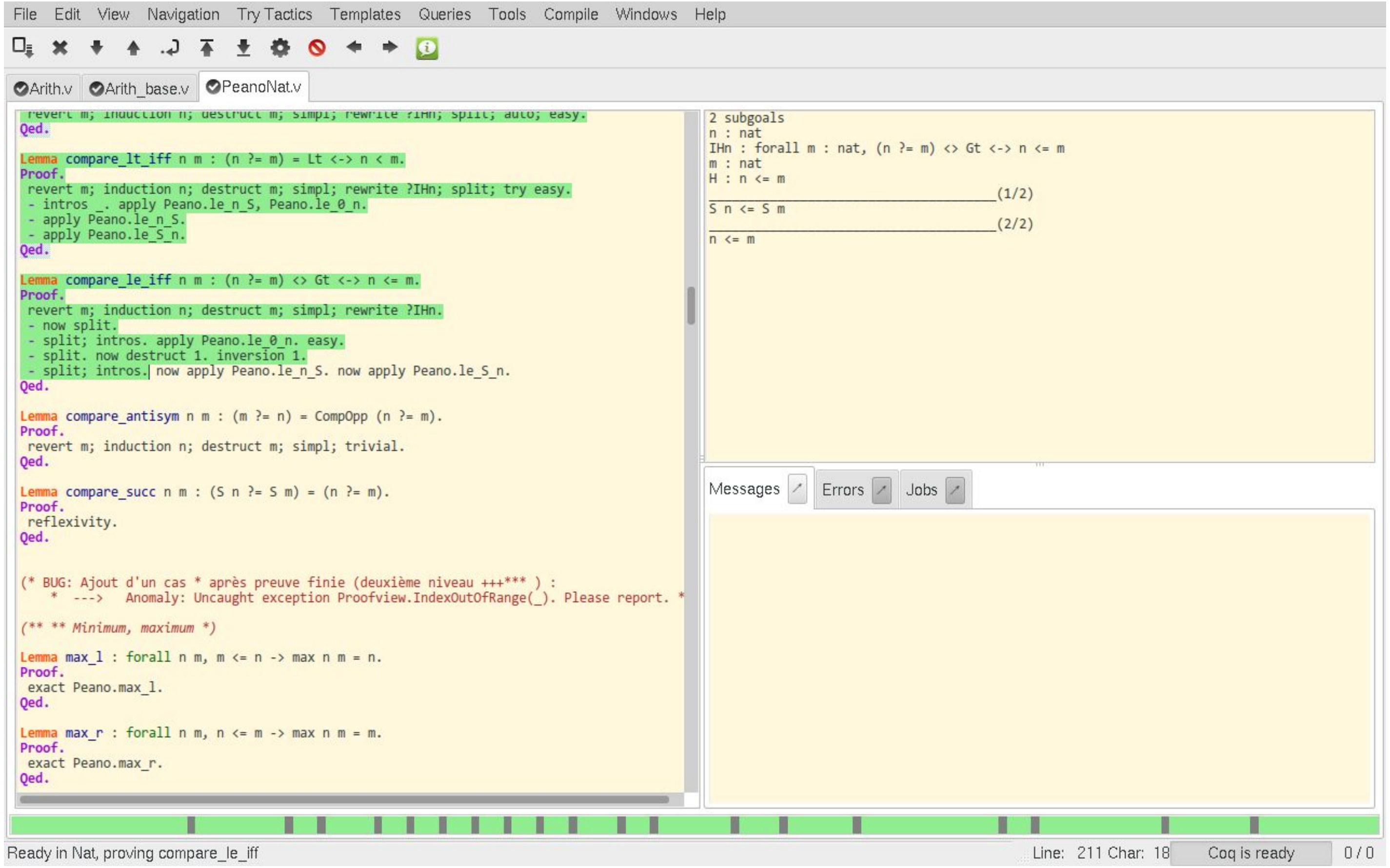
CertiKOS (2016)

- Uses the Coq Proof Assistant

seL4 (2009)

- Uses the Isabelle/HOL theorem prover

Both Coq and Isabelle/HOL are *interactive* theorem provers!



Source: https://upload.wikimedia.org/wikipedia/commons/7/74/Coq_8.5_stdlib_proof.png

Toolchain: Hyperkernel

- Uses the Z3 SMT (satisfiability modulo theories) solver
- Fully automated “push-button” verification using *symbolic execution*
- Implemented on the xv6 educational kernel

Hyperkernel development flow

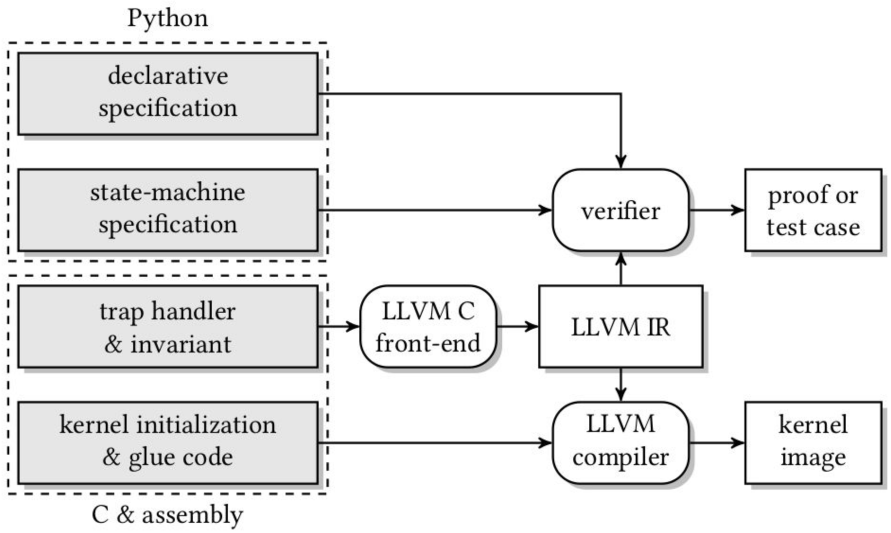


Figure 3: The Hyperkernel development flow. Rectangular boxes denote source, intermediate, and output files; rounded boxes denote compilers and verifiers. Shaded boxes denote files written by programmers.

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

Finitizing xv6 kernel interface

xv6?

- Educational Unix-like kernel from MIT
- Implements Unix V6 on modern systems
- POSIX-like kernel interface

xv6 vs. Hyperkernel Interfaces

xv6

1. syscall semantics require writing *loop invariants* which are slow to verify
2. Kernel pointers difficult to reason about
3. C is difficult to model due to undefined behavior

Hyperkernel

1. Finite syscall interfaces are much faster to verify
2. Separate user and kernel memory
3. Verify LLVM IR instead of C

Example: non-finite dup

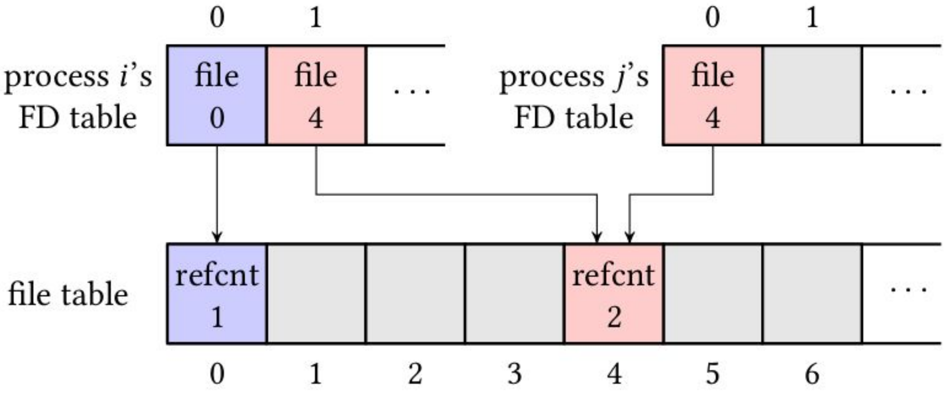


Figure 4: Per-process file descriptor (FD) tables and the system-wide file table.

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

Example: non-finite dup

```
dup(oldfd):
    newfd := 0
    while ft[newfd] is used:
        newfd++
    # copy oldfd to newfd
```

- New FD is the *first unused* FD in the system-wide FD table
- syscall execution time only bounded by size of table

Example: finite dup

```
dup(oldfd, newfd):
    if newfd is unused:
        # copy oldfd to newfd
```

- Only checks the user-provided `newfd`
- Runs in constant time no matter the state of the FD table

Specifications

Specifications

- Use *state-machine specifications* to describe behavior of the kernel
    a. definition of abstract kernel state
    b. definition of trap handlers

Abstract kernel state

- Kernel state definitions written in Python
  - uses fixed-width integers and maps

```
class AbstractKernelState(object):
  current       = PidT()
  proc_fd_table = Map((PidT, FdT), FileT)
  proc_nr_fds   = RefcntMap(PidT, SizeT)
  file_nr_fds   = RefcntMap(FileT, SizeT)
  ...
```

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

State Transitions via Trap Handlers

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

- Define specifications of trap handlers (e.g. syscalls) using Python

```python
def spec_dup(state, oldfd, newfd):
  # state is an instance of AbstractKernelState
  pid = state.current
  # validation condition for system call arguments
  valid = And(
    # oldfd is in [0, NR_FDS)
    oldfd >= 0, oldfd < NR_FDS,
    # oldfd refers to an open file
    state.proc_fd_table(pid, oldfd) < NR_FILES,
    # newfd is in [0, NR_FDS)
    newfd >= 0, newfd < NR_FDS,
    # newfd does not refer to an open file
    state.proc_fd_table(pid, newfd) >= NR_FILES,
  )

  # make the new state based on the current state
  new_state = state.copy()
  f = state.proc_fd_table(pid, oldfd)
  # newfd refers to the same file as oldfd
  new_state.proc_fd_table[pid, newfd] = f
  # bump the FD counter for the current process
  new_state.proc_nr_fds(pid).inc(newfd)
  # bump the counter in the file table
  new_state.file_nr_fds(f).inc(pid, newfd)

  return valid, new_state
```

State Transitions via Trap Handlers

1. Validation of arguments
2. State transition (if valid)

```
def spec_dup(state, oldfd, newfd):
  # state is an instance of AbstractKernelState
  pid = state.current
  # validation condition for system call arguments
  valid = And(
    # oldfd is in [0, NR_FDS)
    oldfd >= 0, oldfd < NR_FDS,
    # oldfd refers to an open file
    state.proc_fd_table(pid, oldfd) < NR_FILES,
    # newfd is in [0, NR_FDS)
    newfd >= 0, newfd < NR_FDS,
    # newfd does not refer to an open file
    state.proc_fd_table(pid, newfd) >= NR_FILES,
  )

  # make the new state based on the current state
  new_state = state.copy()
  f = state.proc_fd_table(pid, oldfd)
  # newfd refers to the same file as oldfd
  new_state.proc_fd_table[pid, newfd] = f
  # bump the FD counter for the current process
  new_state.proc_nr_fds(pid).inc(newfd)
  # bump the counter in the file table
  new_state.file_nr_fds(f).inc(pid, newfd)

  return valid, new_state
```

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

Declarative Specifications

- Define high-level correctness properties about the abstract kernel state
- Authors provide a Python library to help programmers specify these properties

```
ForAll([f, pid, fd], Implies(file_nr_fds(f) == 0,
                             proc_fd_table(pid, fd) != f))
```

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

Verification

Theorem 1: Refinement

- Refinement: verifiable transformation of an abstract specification to its implementation
- Done by specifying an *equivalence function* in Python between LLVM IR data structures and abstract kernel state

Theorem 1: Refinement

DEFINITION 1 (SPECIFICATION-IMPLEMENTATION REFINEMENT). *The kernel implementation is a refinement of the state-machine specification if the following holds for each pair of state transition functions f_{spec} and f_{impl}:*

$$\forall s_{spec}, s_{impl}, x.\ s_{spec} \sim_I s_{impl} \Rightarrow f_{spec}(s_{spec}, x) \sim_I f_{impl}(s_{impl}, x)$$

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

Proving Theorem 1

- Z3 tries to prove that the negation is *unsatisfiable*
- f_{spec} — from state-machine specification
- f_{impl}, I — from exhaustive symbolic execution of LLVM IR code

Theorem 2: Crosscutting

- Whether the state-machine specification satisfies the declarative specification
- Done by translating both specs to SMT and checking that the declarative spec holds after each state transition

Theorem 2: Crosscutting

DEFINITION 2 (STATE-MACHINE SPECIFICATION CORRECTNESS). *The state-machine specification satisfies the declarative specification P if the following holds for every state transition f_{spec} starting from state s_{spec} with input x:*

$$\forall s_{spec}, x.\ P(s_{spec}) \Rightarrow P(f_{spec}(s_{spec}, x))$$

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

Proving Theorem 2

- Z3 tries to prove the negation unsatisfiable (as for Theorem 1)
- Verifier computes the SMT encoding of P and f_{spec} from the Python specifications

Test Generation

Two types of bugs:

1. Bugs in implementation
2. Bugs in state-machine specification

If Z3 cannot find any countexamples, the two theorems hold

Benefits of LLVM IR

- Less undefined behaviors (compared to C)
- Retains high-level information e.g. types (compared to x86 assembly)
- Does not include machine-specific details like stack pointer (compared to x86 assembly)

Encoding LLVM IR in SMT

- Many types, instructions map directly
    - n-bit LLVM int -> n-bit SMT bit-vector
    - LLVM add instruction -> SMT bit-vector addition
- LLVM volatile memory accesses need special care
    - each volatile read mapped to new symbolic variable
- Must handle undefined behavior
- Verifier does not support exceptions, int-to-ptr conversions, floats as Hyperkernel does not use them

Hyperkernel Design

Processes through hardware virtualization

- Kernel runs as host, user processes run as ring 0 guests
    - Inspired by Dune
- Allows kernel and userspace to have separate page tables
    - Easier verification
- Exceptions can bypass kernel, delivered straight to userspace
    - Performance benefits
    - Most exception handling done in userspace (except triple faults)

Explicit resource management

- System calls require user to make resource allocation decisions
  - e.g. `dup(oldfd, newfd)`
- Avoids loops in trap handlers
- Implement data structures using arrays instead of linked lists or trees

Hypercalls

- Userspace uses *hypercalls* to invoke the kernel due to virtualization
  - e.g. `vmcall` to call to VM monitor
- Slower than Linux system calls
- Interrupts disabled during a hypercall's execution

Experience

Bugs in xv6

Commit	Description	Preventable?
8d1f9963	incorrect pointer	● verifier
2a675089	bounds checking	● verifier
ffe44492	memory leak	● verifier
aff0c8d5	incorrect I/O privilege	● verifier
ae15515d	buffer overflow	● verifier/boot checker
5625ae49	integer overflow in exec	◐
e916d668	signedness error in exec	◐
67a7f959	alignedness error in exec	◐

Figure 7: Bugs found and fixed in xv6 in the past year and whether they can be prevented in Hyperkernel: ● means the bug can be prevented through the verifier or checkers; and ◐ means the bug can be prevented in the kernel but can happen in user space.

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

Run-time performance

Benchmarks

- syscall: no-op syscall
- fault: time to invoke page fault handler
- appel{1,2}: access to protected pages

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

Benchmark	Linux	Hyperkernel	Hyp-Linux
syscall	125	490	136
fault	2,917	615	722
appel1	637,562	459,522	519,235
appel2	623,062	452,611	482,596

Figure 10: Cycle counts of benchmarks running on Linux, Hyperkernel, and Hyp-Linux (the Linux emulation layer for Hyperkernel).

Syscall vs. Hypercall performance

Model	Microarchitecture	Syscall	Hypercall
Intel			
Xeon X5550	Nehalem (2009)	72	961
Xeon E5-1620	Sandy Bridge (2011)	72	765
Core i7-3770	Ivy Bridge (2012)	74	760
Xeon E5-1650 v3	Haswell (2013)	74	540
Core i5-6600K	Skylake (2015)	79	568
Core i7-7700K	Kaby Lake (2016)	69	497
AMD			
Ryzen 7 1700	Zen (2017)	64	697

Figure 11: Cycle counts of syscalls and hypercalls on x86 processors; each result averages 50 million trials.

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

Limitations and Future Work

Limitations

- xv6 lacks features
  - Threading
  - Copy-on-write fork
  - Shared pages
  - Unix permissions
- Hyperkernel cannot fully implement POSIX
  - `fork`, `exec`, `mmap`, etc. are non-finite and difficult to write specs for
- Hyperkernel only verifies LLVM IR
  - Correctness guarantees do not extend to C source code or final binary
  - Cannot model machine details such as the stack
- Lots of trusted code
  - kernel initialization code
  - glue code

Limitations

- Some correctness of syscalls pushed to userspace
    - e.g. process creation

Class Comments

- Using this approach for more featureful kernels like Linux
- Where does the specification come from?
- Verifying LLVM IR does not extend guarantees to C or binary
- Relies on small constraint model, potentially increase code coverage with static symbolic execution
- Trusting hardware and init code is an issue
- Hyperkernel better suited for simpler devices e.g. IoT
- Yggdrasil as motivation