Hyperkernel: Push-Button Verification of an OS Kernel

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson,
James Bornholt, Emina Torlak, and Xi Wang
University of Washington
{lukenels,helgi,kaiyuanz,dgj16,bornholt,emina,xi}@cs.washington.edu

ABSTRACT

This paper describes an approach to designing, implementing, and formally verifying the functional correctness of an OS kernel, named Hyperkernel, with a high degree of proof automation and low proof burden. We base the design of Hyperkernel's interface on xv6, a Unix-like teaching operating system. Hyperkernel introduces three key ideas to achieve proof automation: it finitizes the kernel interface to avoid unbounded loops or recursion; it separates kernel and user address spaces to simplify reasoning about virtual memory; and it performs verification at the LLVM intermediate representation level to avoid modeling complicated C semantics.

We have verified the implementation of Hyperkernel with the Z3 SMT solver, checking a total of 50 system calls and other trap handlers. Experience shows that Hyperkernel can avoid bugs similar to those found in xv6, and that the verification of Hyperkernel can be achieved with a low proof burden.

ACM Reference Format:
Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3132747.3132748

1 INTRODUCTION

The OS kernel is one of the most critical components of a computer system, as it provides essential abstractions and services to user applications. For instance, the kernel enforces isolation between processes, allowing multiple applications to safely share resources such as CPU and memory. Consequently, bugs in the kernel have serious implications for correctness and security, from causing incorrect behavior in individual applications to allowing malicious applications to compromise the entire system [12, 14, 50, 55].

Previous research has applied formal verification to eliminate entire classes of bugs within OS kernels, by constructing a machine-checkable proof that the behavior of an implementation adheres to its specification [25, 34, 69]. But these impressive achievements come with a non-trivial cost. For example, the functional correctness proof of the seL4 kernel took roughly 11 person years for 10,000 lines of C code [35].

This paper explores a push-button approach to building a provably correct OS kernel with a low proof burden. We take as a starting point the xv6 teaching operating system [17], a modern re-implementation of the Unix V6 for x86. Rather than using interactive theorem provers such as Isabelle [54] or Coq [16] to manually write proofs, we have redesigned the xv6 kernel interface to make it amenable to automated reasoning using satisfiability modulo theories (SMT) solvers. The resulting kernel, referred to as the Hyperkernel in this paper, is formally verified using the Z3 SMT solver [19].

A key challenge in verifying Hyperkernel is one of interface design, which needs to strike a balance between usability and proof automation. On one hand, the kernel maintains a rich set of data structures and invariants to manage processes, virtual memory, and devices, among other resources. As a result, the Hyperkernel interface needs to support specification and verification of high-level properties (e.g., process isolation) that provide a basis for reasoning about the correctness of user applications. On the other hand, this interface must also be implementable in a way that enables fully automated verification of such properties with an SMT solver.

A second challenge arises from virtual memory management in kernel code. Figure 1 shows a typical address space layout on x86: both the kernel and user space reside in the same virtual address space, with the kernel taking the upper half and leaving the lower half to user space [46]. The kernel virtual memory is usually not an injective mapping—writing to one kernel memory address can change the value at another address, as two virtual addresses may both map to the same physical address. Therefore, reasoning about kernel data structures requires reasoning about the virtual-to-physical mapping. This reasoning task is further complicated by the fact that kernel code can change the virtual-to-physical mapping during execution. Proving properties about kernel code is particularly difficult in such a setting [37, 38].

SOSP '17, October 28, 2017, Shanghai, China
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132748

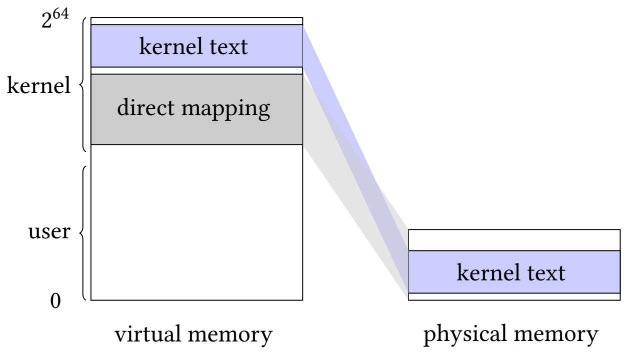

Figure 1: A simplified memory layout of Linux on x86-64: the kernel and user space are mapped to the upper half and lower half of the virtual address space, respectively. The ABI recommends the kernel text to be mapped to the top 2 GiB, as required by the kernel code model [52]; the kernel also has a direct mapping of all physical memory.

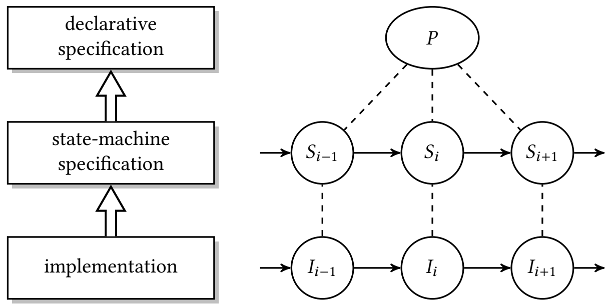

Figure 2: An overview of Hyperkernel verification. S_i and I_i denote states of the corresponding layers; solid arrows denote state transitions. P denotes a crosscutting property that holds during every state transition.

A final challenge is that Hyperkernel, like many other OS kernels, is written in C, a programming language that is known to complicate formal reasoning [26, 39, 53]. It is notably difficult to accurately model the C semantics and reason about C programs due to low-level operations such as pointer arithmetic and memory access. In addition, the C standard is intentionally underspecified, allowing compilers to exploit undefined behavior in order to produce efficient code [41, 67]. Such subtleties have led some researchers to conclude that "there is no C program for which the standard can guarantee that it will not crash" [40].

Hyperkernel addresses these challenges with three ideas. First, its kernel interface is designed to be *finite*: all of the handlers for system calls, exceptions, and interrupts (collectively referred to as *trap handlers* in this paper) are free of unbounded loops and recursion, making it possible to encode and verify them using SMT. Second, Hyperkernel runs in a *separate* address space from user space, using an identity mapping for the kernel; this simplifies reasoning about kernel code. To efficiently realize this separation, Hyperkernel makes use of x86 virtualization support provided by Intel VT-x and AMD-V: the kernel and user processes run in root (host) and non-root (guest) modes, respectively, using separate page tables. Third, Hyperkernel performs verification at the level of the LLVM intermediate representation (IR) [42], which has much simpler semantics than C while remaining sufficiently high-level to avoid reasoning about machine details.

The kernel interface of Hyperkernel consists of 50 trap handlers, providing support for processes, virtual memory, file descriptors, devices, inter-process communication, and scheduling. We have verified the correctness of this interface in two steps, as shown in Figure 2. First, we have developed a specification of trap handlers in a *state-machine* style, describing the intended behavior of the implementation. Second, to improve the confidence in the correctness of the state-machine specification, we have further developed a higher-level specification in a *declarative* style. The declarative specification describes "end-to-end" crosscutting properties that the state-machine specification must satisfy [60], such as "a process can write to pages only owned by itself." Such properties are more intuitive and easier to review. Using the Z3 SMT solver, verification finishes within about 15 minutes on an 8-core machine.

The current prototype of Hyperkernel runs on a uniprocessor system; verifying multiprocessor support is beyond the scope of this paper. We choose not to verify the kernel initialization and glue code (e.g., assembly for register save and restore), instead relying on a set of custom checkers to improve confidence in their correctness.

To demonstrate the usability of the kernel interface, we have ported xv6 user programs to Hyperkernel, including utilities and a shell. We have also ported the xv6 journaling file system and the lwIP networking stack, both running as user-space processes. We have developed several applications, including a Linux binary emulator and a web server that can host the Git repository of this paper.

In summary, this paper makes two main contributions: a push-button approach to building a verified OS kernel, and a kernel interface design amenable to SMT solving. The careful design of the kernel interface is key to achieving a high degree of proof automation—naïvely applying the Hyperkernel approach to verifying an existing kernel is unlikely to scale. We chose xv6 as a starting point as it provides classic Unix abstractions, with the final Hyperkernel interface, which is amenable to automated verification, resembling an exokernel [23, 33]. We hope that our experience can provide inspiration for designing other "push-button verifiable" interfaces.

The rest of this paper is organized as follows. §2 gives an overview of the verification process. §3 presents formal definitions and verification details. §4 describes the design and implementation of Hyperkernel and user-space libraries. §5 discusses checkers as extensions to our verifier. §6 reports on verification and runtime performance of Hyperkernel. §7 relates Hyperkernel to prior work. §8 concludes.

2 OVERVIEW

This section illustrates the Hyperkernel development workflow by walking through the design, specification, and verification of one system call.

As shown in Figure 3, to specify the desired behavior of a system call, programmers write two forms of specifications: a detailed, state-machine specification for functional correctness, and a higher-level, declarative specification that is more intuitive for manual review. Both specifications are expressed in Python, which we choose due to its simple syntax and user-friendly interface to the Z3 SMT solver. Programmers implement a system call in C. The verifier reduces both specifications (in Python) and the implementation (in LLVM IR compiled from C) into an SMT query, and invokes Z3 to perform verification. The verified code is linked with unverified (trusted) code to produce the final kernel image.

An advantage of using an SMT solver is its ability to produce a test case if verification fails, which we find useful for pinpointing and fixing bugs. For instance, if there is any bug in the C code, the verifier generates a concrete test case, including the kernel state and system call arguments, to describe how to trigger the bug. Similarly, the verifier shows the violation if there is any inconsistency between the two forms of specifications.

We make the following two assumptions in this section. First, the kernel runs on a uniprocessor system with interrupts disabled. Every system call is therefore atomic and runs to completion. Second, the kernel is in a separate address space from user space, using an identity mapping for virtual memory. These assumptions are explained in detail in §3.

2.1 Finite interfaces

We base the Hyperkernel interface on existing specifications (such as POSIX), making adjustments where necessary to aid push-button verification. In particular, we make adjustments to keep the kernel interface *finite*, by ensuring that the semantics of every trap handler is expressible as a set of traces of bounded length. To make verification scalable, these bounds should be small constants that are independent of system parameters (e.g., the maximum number of file descriptors or pages).

To illustrate the design of finite interfaces, we show how to finitize the POSIX specification of the `dup` system call

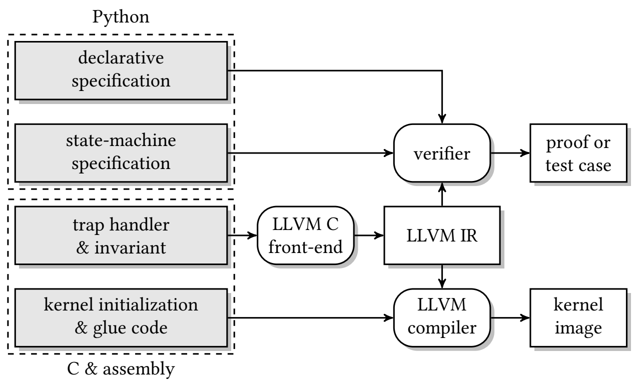

Figure 3: The Hyperkernel development flow. Rectangular boxes denote source, intermediate, and output files; rounded boxes denote compilers and verifiers. Shaded boxes denote files written by programmers.

for inclusion in Hyperkernel. In a classic Unix design, each process maintains a *file descriptor (FD) table*, with each slot in this table referring to an entry in a system-wide *file table*. Figure 4 shows two example FD tables, for processes *i* and *j*, along with a system-wide file table. The slot FD 0 in process *i*'s table refers to the file table entry 0, and both process *i*'s FD 1 and process *j*'s FD 0 refer to the same file table entry 4. To correctly manage resources, the file table maintains a reference counter for each entry: entry 4's counter is 2 as it is referred to by two FDs.

The POSIX semantics of `dup(oldfd)` is to create "a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor" [47]. For example, invoking `dup(0)` in process *j* would return FD 1 referring to file table entry 4, and increment that entry's reference counter to 3.

We consider the POSIX semantics of the `dup` interface as non-finite. To see why, observe that the lowest-FD semantics, although rarely needed in practice [15], requires the kernel implementation to check that every slot lower than the new chosen FD is already occupied. As a result, allocating the lowest FD requires a trace that grows with the size of the FD table—i.e., trace length cannot be bounded by a small constant that is independent of system parameters. This lack of a small finite bound means that the verification time of `dup` would increase with the size of the FD table.

Hyperkernel finitizes `dup` by changing the POSIX interface to `dup(oldfd, newfd)`, which requires user space to choose a new FD number. To implement this interface, the kernel simply checks whether a given `newfd` is unused. Such a check requires a small, constant number of operations, irrespective of the size of the FD table. This number puts an upper bound on the length of any trace that a call to `dup(oldfd, newfd)` can generate; the interface is therefore finite, enabling scalable verification.

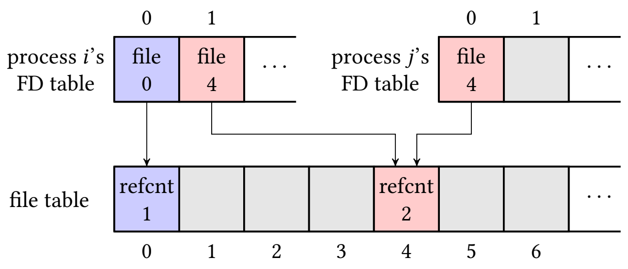

Figure 4: Per-process file descriptor (FD) tables and the system-wide file table.

We emphasize two benefits of a finite interface. First, although it is possible to verify the POSIX dup using SMT if the FD table is small (by simply checking all possible table sizes), this would not scale for resources with large parameters (e.g., pages). Therefore, we apply the finite-interface design to all of Hyperkernel's trap handlers. Second, our definition of finite interfaces does not bound the size of kernel state—only trace length. The kernel state can thus include arbitrarily many file descriptors or pages, as long as each trap handler accesses only a constant number of them, independent of the size of the state.

2.2 Specifications

Given a finite interface, the programmer describes the desired behavior of the kernel by providing a *state-machine specification*. This specification consists of two parts: a definition of abstract kernel state, and a definition of trap handlers (e.g., system calls) in terms of abstract state transitions. In addition to a state-machine specification, the programmer can optionally provide a *declarative specification* of the high-level properties that the state-machine specification should satisfy. The Hyperkernel verifier will check that these high-level properties are indeed satisfied, helping increase the programmer's confidence in the correctness of the state-machine specification. This section develops both forms of specification for the finite dup interface.

Abstract kernel state. Programmers define the abstract kernel state using fixed-width integers and maps, as follows:

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

This snippet defines four components of the abstract state:

- current is the current running process's identifier (PID);
- proc_fd_table represents per-process FD tables, mapping a PID and an FD to a file;
- proc_nr_fds maps a PID to the number of FDs used by that process; and
- file_nr_fds maps a file to the number of FDs (across all processes) that refer to that file.

The types PidT, FdT, FileT, and SizeT correspond to SMT fixed-width integers (i.e., bit-vectors). The Map constructor creates an uninterpreted function [20], which maps one or more domain types to a range type. RefcntMap is a special map for reference counting.

State-transition specification. The specification of most system calls, including dup, follows a common pattern: it validates system call arguments and transitions the kernel to the next state if validation passes, returning zero as the result; otherwise, the system call returns an error code and the kernel state does not change. Each system call specification provides a validation condition and the new state, as follows:

```
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
```

The specification of dup takes as input the current abstract kernel state and its arguments (oldfd and newfd). Given these inputs, it returns a validation condition and the new state to which the kernel will transition if the validation condition is true. And is a built-in logical operator; NR_FDS and NR_FILES are the size limits of the FD table and the file table, respectively; and inc bumps a reference counter by one, taking a parameter to specify the newly referenced resource (used to formulate reference counting; see §3.3). For simplicity, we do not model error codes here.

Declarative specification. The state-machine specification of dup is abstract: it does not have any undefined behavior as in C, or impose implementation details like data layout in memory. But it still requires extra care; for example, the programmer needs to correctly modify reference counters in the file table when specifying dup. To improve confidence in its correctness, we also develop a higher-level declarative specification [60] to better capture programmer intuition about kernel behavior, in the form of a conjunction of crosscutting properties that hold across all trap handlers.

Consider a high-level correctness property for reference counting in the file table: if a file's reference count is zero, there must be no FD referring to the file. Programmers can specify this property as follows:

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

Here, ForAll and Implies are built-in logical operators. Every trap handler, including dup, should maintain this property.

More generally, every trap handler should maintain that each file f's reference count is equal to the total number of per-process FDs that refer to f:

$$\texttt{file\_nr\_fds}(f) = |\{(pid,\ fd) \mid \texttt{proc\_fd\_table}(pid,\ fd) = f\}|$$

We provide a library to simplify the task of expressing such reference counting properties, further explained in §3.3.

This declarative specification captures the intent of the programmer, ensuring that the state-machine specification—and therefore the implementation—satisfies desirable crosscutting properties. For reference counting, even if both the state-machine specification and the implementation failed to correctly update a reference counter, the declarative specification would expose the bug. §6.1 describes one such bug.

2.3 Implementation

The C implementation of dup(oldfd, newfd) in Hyperkernel closely resembles that in xv6 [17] and Unix V6 [48]. The key difference is that rather than searching for an unused FD, the code simply checks whether a given newfd is unused.

Briefly, the Hyperkernel implementation of dup uses the following data structures:

- a global integer current, representing the current PID;
- a global array procs[NR_PROCS] of struct proc objects, representing at most NR_PROCS processes;
- each struct proc contains an array ofile[NR_FDS] mapping file descriptors to files; and
- a global array files[NR_FILES] representing the file table, mapping files to struct file objects.

The implementation copies the file referred to by oldfd (i.e., procs[current].ofile[oldfd]) into the unused newfd (i.e., procs[current].ofile[newfd]), and increments the corresponding reference counters. It also checks the values of oldfd and newfd to avoid buffer overflows, as they are supplied by user space and used to index into the ofile array. We omit the full code here due to space limitation.

Unlike previous kernel verification projects [34], we have fewer restrictions on the use of C, as the verification will be performed at the LLVM IR level. For instance, programmers are allowed to use goto or fall-through switch statements. See §3 for details.

Representation invariant. The kernel explicitly checks the validity of values from user space (such as system call arguments), as they are untrusted. But the validity of values within the kernel is often implicitly assumed. For example, consider the global variable current, which holds the PID of the current running process. The implementation of the dup system call uses current to index into the array procs. To check (rather than assume) that this access does not cause a buffer overflow, the Hyperkernel programmer has two options: a dynamic check or a static check.

The dynamic check involves inserting the following test into dup (and every other system call that uses current):

```
if (current > 0 && current < NR_PROCS) { ... }
```

The downside is that this check will always evaluate to true at run time if the kernel is correctly implemented, unnecessarily bloating the kernel and wasting CPU cycles.

The static check performs such tests at verification time. To do so, programmers write the same range check of current, but in a special check_rep_invariant() function, which describes the *representation invariant* of kernel data structures. The verifier will try to prove that every trap handler maintains the representation invariant.

2.4 Verification

The verification of Hyperkernel proves two main theorems:

Theorem 1 (Refinement). *The kernel implementation is a refinement of the state-machine specification.*

Theorem 2 (Crosscutting). *The state-machine specification satisfies the declarative specification.*

Proving Theorem 1 requires programmers to write an equivalence function (in Python) to establish the correspondence between the kernel data structures in LLVM IR (compiled from C) and the abstract kernel state. This function takes the form of a conjunction of constraints that relate variables in the implementation to their counterparts in the abstract state. For example, consider the following equivalence constraint:

```
llvm_global('@current') == state.current
```

On the left-hand side, llvm_global is a helper function that looks up a symbol current in the LLVM IR (@ indicates a global symbol in LLVM), which refers to the current PID in the implementation; on the right-hand side, `state.current` refers to the current PID in the abstract state, as defined in §2.2. Other pairs of variables are similarly constrained.

Using the equivalence function, the verifier proves Theorem 1 as follows: it translates both the state-machine specification (written in Python) and the implementation (in LLVM IR) into SMT, and checks whether they move in lock-step for every state transition. The theorem employs a standard definition of refinement (see §3): assuming that both start in equivalent states and the representation invariant holds, prove that they transition to equivalent states and the representation invariant still holds.

To prove Theorem 2, that the state-machine specification satisfies the declarative specification (both written in Python), the verifier translates both into SMT and checks that the declarative specification holds after each state transition assuming that it held beforehand.

Test generation. The verifier can find the following classes of bugs if it fails to prove the two theorems:

- bugs in the implementation (undefined behavior or violation of the state-machine specification), and
- bugs in the state-machine specification (violation of the declarative specification).

In these cases the verifier attempts to produce a concrete test case from the Z3 counterexample to help debugging.

For example, if the programmer forgot to validate the system call parameter `oldfd` in `dup`, the verifier would output a stack trace along with a concrete `oldfd` value that causes an out-of-bounds access in the FD table.

As another example, if the programmer forgot to increment the reference counter in the file table in the `dup` implementation, the verifier would highlight the clause being violated in the state-machine specification, along with the following (simplified) explanation:

```
# kernel state:
[oldfd = 1, newfd = 0, current = 32,
 proc_fd_table = [(32, 1) -> 1, else -> -1]
 file_nr_fds = [1 -> 1, else -> 0],
 @files->struct.file::refcnt = [1 -> 1, else -> 0]
 ...]

# before (assumption):
ForAll([f],
  @files->struct.file::refcnt(f) == file_nr_fds(f))

# after (fail to prove):
ForAll([f]
  @files->struct.file::refcnt(f) == If(
    f == proc_fd_table(current, oldfd),
    file_nr_fds(f) + 1,
    file_nr_fds(f)))
```

This output says that a bug can be triggered by invoking `dupfd(1, 0)` within the process of PID 32. The kernel state before the system call is the following: PID 32 is the current running process; its FD 1 points to file 1 (with reference counter 1); other FDs and file table entries are empty. The two `ForAll` statements highlight the offending states before and after the system call, respectively. Before the call, the specification and implementation states are equivalent. After the system call, the counter in the specification is correctly updated (i.e., file 1's counter is incremented by one in `file_nr_fds`); however, the counter remains unchanged in the implementation (i.e., `@files->struct.file::refcnt`), which breaks the equivalence function and so the proof fails.

Theorems. The two theorems hold if Z3 cannot find any counterexamples. In particular, Theorem 1 guarantees that the verified part of the kernel implementation is free of low-level bugs, such as buffer overflow, division by zero, and null-pointer dereference. It further guarantees functional correctness—that each system call implementation satisfies the state-machine specification.

Theorem 2 guarantees that the state-machine specification is correct with respect to the declarative specification. For example, the declarative specification in §2.2 requires that file table reference counters are consistently incremented and decremented in the state-machine specification.

Note that the theorems do not guarantee the correctness of kernel initialization and glue code, or the resulting kernel binary. §5 will introduce separate checkers for the unverified parts of the implementation.

2.5 Summary

Using the `dup` system call, we have illustrated how to design a finite interface, write a state-machine specification and a higher-level declarative specification, implement it in C, and verify the correctness in Hyperkernel. The proof effort is low thanks to the use of Z3 and the kernel interface design. For `dup`, the proof effort consists mainly of writing the specifications, one representation invariant on `current` (or adding dynamic checks instead), and the equivalence function.

The trusted computing base (TCB) includes the specifications, the theorems (including the equivalence function), kernel initialization and glue code, the verifier, and the dependent verification toolchain (i.e., Z3, Python, and LLVM). The C frontend to LLVM (including the LLVM IR optimizer) is not trusted. Hyperkernel also assumes the correctness of hardware, such as CPU, memory, and IOMMU.

3 THE VERIFIER

To prove Hyperkernel's two correctness theorems, the verifier encodes kernel properties in SMT for automated verification. To make verification scalable, the verifier restricts its

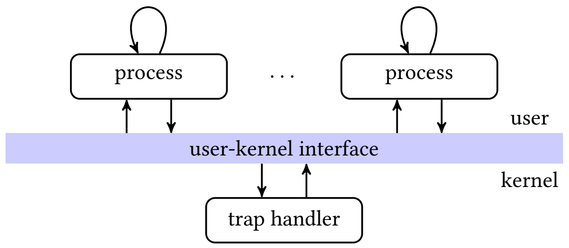

Figure 5: State transitions in Hyperkernel. At each step process execution may either stay in user space or trap into the kernel due to system calls, exceptions, or interrupts. Each trap handler in the kernel runs to completion with interrupt disabled.

use of SMT to an effectively decidable fragment of first-order logic. This section describes how we use this restriction to guide the design of the formalization.

We first present our model of the kernel behavior as a state machine (§3.1), followed by the details of the verification process. In particular, to verify the C implementation against the state-machine specification, the verifier translates the semantics of the LLVM IR into an SMT expression (§3.2). To check the state-machine specification against the declarative specification (e.g., the correctness of reference counters), it encodes crosscutting properties in a way that is amenable to SMT solving (§3.3).

3.1 Modeling kernel behavior

The verifier follows a standard way of modeling a kernel's execution as a state machine [36]. As shown in Figure 5, a state transition can occur in response to either trap handling or user-space execution (without trapping into the kernel). By design, the execution of a trap handler in Hyperkernel is *atomic*: it traps from user space into the kernel due to system calls, exceptions, or interrupts, runs to completion, and returns to user space. This atomicity simplifies verification by ruling out interleaved execution, allowing the verifier to reason about each trap handler in its entirety and independently.

As mentioned earlier, Hyperkernel runs on a uniprocessor system. However, even in this setting, ensuring the atomic execution of trap handlers requires Hyperkernel to sidestep concurrency issues that arise from I/O devices, namely, interrupts and direct memory access (DMA), as follows.

First, the kernel executes trap handlers with interrupts disabled, postponing interrupts until the execution returns to user space (which will trap back into the kernel). By doing so, each trap handler runs to completion in the kernel.

Second, since devices may asynchronously modify memory through DMA, the kernel isolates their effects by restricting DMA to a dedicated memory region (referred to as *DMA pages*); this isolation is implemented through mechanisms such as Intel's VT-d Protected Memory Regions [29] and AMD's Device Exclusion Vector [4] configured at boot time. In addition, the kernel conservatively considers DMA pages *volatile* (see §3.2), where memory reads return arbitrary values. In doing so, a DMA write that occurs during kernel execution is effectively equivalent to a no-op with respect to the kernel state, removing the need to explicitly model DMA.

With this model, we now define kernel correctness in terms of state-machine refinement. Formally, we denote each state transition (e.g., trap handling) by a transition function f that maps the current state s and input x (e.g., system call arguments) to the next state $f(s, x)$. Let f_{spec} and f_{impl} be the transition functions for the specification and implementation of the same state transition, respectively. Let I be the representation invariant of the implementation (§2.3). Let $s_{spec} \sim s_{impl}$ denote that specification state s_{spec} and implementation state s_{impl} are equivalent according to the programmer-defined equivalence function (§2.4). We write $s_{spec} \sim_I s_{impl}$ as a shorthand for $I(s_{impl}) \wedge (s_{spec} \sim s_{impl})$, which states that the representation invariant holds in the implementation and both states are equivalent. With this notation, we define refinement as follows:

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}:*

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

To prove kernel correctness (Theorem 1), the verifier computes the SMT encoding of f_{spec} and f_{impl} for each transition function f, as well as the representation invariant I (which is the same for all state transitions). The verifier then asks Z3 to prove the validity of the formula in Definition 1 by showing its negation to be unsatisfiable. The verifier computes f_{spec} by evaluating the state-machine specification written in Python. To compute f_{impl} and I, it performs exhaustive (all-paths) symbolic execution over the LLVM IR of kernel code. If Z3 finds the query unsatisfiable, verification succeeds. Otherwise, if Z3 returns a counterexample, the verifier constructs a test case (§2.4).

Proving crosscutting properties (Theorem 2) is simpler. Since a declarative specification defines a predicate P over the abstract kernel state, the verifier checks whether P holds during each transition of the state-machine specification. More formally:

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:*

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

To prove a crosscutting property P, the verifier computes the SMT encoding of P and f_{spec} from the respective specifications (both in Python), and invokes Z3 on the negation of the formula in Definition 2. As before, verification succeeds if Z3 finds this query unsatisfiable.

Note that the verifier assumes the correctness of kernel initialization, leaving the validation to a boot checker (see §5). Specifically, for Theorem 1, it assumes the initial state of the implementation satisfies the representation invariant I; for Theorem 2, it assumes the initial state of the state-machine specification satisfies the predicate P.

3.2 Reasoning about LLVM IR

LLVM IR is a code representation that has been widely used for building compilers and bug-finding tools (e.g., KLEE [11]). We choose it as our verification target for two reasons. One, its semantics is simple compared to C and exhibits fewer undefined behaviors. Two, compared to x86 assembly, it retains high-level information, such as types, and does not include machine-specific details like the stack pointer.

To construct an SMT expression for each trap handler, the verifier performs symbolic execution over its LLVM IR. Specifically, the symbolic execution uses the *self-finitization* strategy [65]: it simply unrolls all the loops and exhaustively traverses every code branch. In doing so, the verifier assumes that the implementation of every trap handler is finite. If not, symbolic execution diverges and verification fails.

The symbolic execution consists of two steps: it emits checks to preclude any undefined behavior in the LLVM IR, and maps LLVM IR into SMT, as detailed next.

Precluding undefined behavior. The verifier must prove that each trap handler is free of undefined behavior. There are three types of undefined behavior in LLVM IR: immediate undefined behavior, undefined values, and poison values [49]. The verifier handles each case conservatively, as follows:

- Immediate undefined behavior indicates errors, such as division by zero. The verifier emits a side check to ensure the responsible conditions do not occur (e.g., divisors must be non-zero).
- Undefined values are a form of deferred undefined behavior, representing arbitrary bits (e.g., from uninitialized memory reads). The verifier represents undefined values with fresh symbolic variables, which may take any concrete value.
- Poison values are like undefined values, but trigger immediate undefined behavior if they reach side-effecting operations. They were introduced to enable certain optimizations, but are known to have subtle semantics; there are ongoing discussions in the LLVM community on removing them (e.g., see Lee et al. [43]). The verifier takes a simple approach: it guards LLVM instructions that may produce poison values with conditions that avoid poison, effectively treating them as immediate undefined behavior.

Encoding LLVM IR in SMT. With undefined behavior precluded, it is straightforward for the verifier to map LLVM types and instructions to SMT. For instance, an n-bit LLVM integer maps to an n-bit SMT bit-vector; LLVM's `add` instruction maps to SMT's bit-vector addition; and regular memory accesses map to uninterpreted function operations [20]. Volatile memory accesses (e.g., DMA pages and memory-mapped device registers), however, require special care: the verifier conservatively maps a volatile read to a fresh symbolic variable that may take any concrete value.

The verifier also allows programmers to provide an abstract model in SMT for inline assembly code. This model is trusted and not verified. Hyperkernel currently uses this support for modeling TLB flush instructions.

The verifier supports a substantial subset of LLVM IR. It does not support exceptions, integer-to-pointer conversions, floating point types, or vector types (e.g., for SSE instructions), as they are not used by Hyperkernel.

3.3 Encoding crosscutting properties

To achieve scalable verification, the verifier restricts the use of SMT to an effectively decidable fragment of first-order logic. The emitted encoding from both LLVM IR and the state-machine specification consists largely of quantifier-free formulas in decidable theories (i.e., bit-vectors and equality with uninterpreted functions).

The exception to this encoding discipline is the use of quantifiers in the high-level, declarative specification. In particular, we use quantifiers to specify properties about two common resource management patterns in Unix-like kernels: that a resource is exclusively owned by one object, and that a shared resource is consistently reference-counted. While such quantified formulas are decidable, encoding them in SMT requires caution—naïve encodings can easily cause the solver to enumerate the search space and fail to terminate within a reasonable amount of time. We next describe SMT encodings that scale well in practice.

Specifying exclusive ownership. Exclusive ownership properties are common in kernel resource management. For example, each process has its own separate address space, and so the page table root of a process must be exclusively owned by a single process. In general, such properties can be expressed in the following form:

$$\forall o, o' \in O.\ own(o) = own(o') \Rightarrow o = o'$$

Here, O is a set of kernel objects (e..g, processes) that can refer to resources such as pages, and *own* is a function that maps an object to a resource (e.g., the page frame number of the page table root of a process). This encoding, however, does not work well in practice.

For effective verification, we reformulate the naïve encoding in a standard way [20] by observing that the *own* function must be injective due to exclusive ownership. In particular, there exists an inverse function *owned-by* such that:

$$\forall o \in O.\ \textit{owned-by}(\textit{own}(o)) = o$$

The verifier provides a library using this encoding: it asks programmers to provide the inverse function, which usually already exists in the state-machine specification (e.g., a map from a page to its owner process), hiding the rest of the encoding details from programmers.

Specifying reference-counted shared ownership. Reference-counted shared ownership is a more general resource management pattern. For example, two processes may refer to the same file through their FDs. The corresponding crosscutting property (described in §2.2) is that the reference count of a file must equal the number of per-process FDs that refer to this file. As another example, the number of children of a process p must equal the number of processes that designate p as their parent.

In general, such properties require the reference count of a shared kernel resource (e.g., a file) to equal the size of the set of kernel objects (e.g., per-process FDs) that refer to it. Formally, verifying such a property involves checking the validity of the formula:

$$\forall r.\ \textit{refcnt}(r) = |\{o \in O \mid \textit{own}(o) = r\}|$$

Here, *refcnt* is a function that maps a resource r to its reference count; O is the set of kernel objects that can hold references to such resources; and *own* maps an object o in O to the resource it refers to.

We encode the above property for scalable verification by observing that if an object r has reference count $\textit{refcnt}(r)$, there must be a way to permute the elements of O such that exactly the first $\textit{refcnt}(r)$ objects in O refer to r.

In particular, the verifier encodes the reference counting property in two parts. First, for each resource r, a permutation π orders the objects of O so that only the first $\textit{refcnt}(r)$ objects refer to the resource r:

$$\begin{aligned}&\forall r.\ \forall 0 \le i < |O|.\\ &\quad \textit{own}(\pi(r,i)) = r \Rightarrow i < \textit{refcnt}(r) \wedge\\ &\quad \textit{own}(\pi(r,i)) \ne r \Rightarrow i \ge \textit{refcnt}(r)\end{aligned}$$

Second, π must be a valid permutation (i.e., a bijection), so there exists an inverse function π^{-1} such that:

$$\begin{aligned}\forall r.\ &\left[\forall 0 \le i < |O|.\ \pi^{-1}(r, \pi(r,i)) = i\right] \wedge\\ &\left[\forall o \in O.\ \pi(r, \pi^{-1}(r,o)) = o\right]\end{aligned}$$

A library hides these details from programmers (see §2.2).

4 THE HYPERKERNEL

This section describes how to apply the finite-interface design and make Hyperkernel amenable to automated verification. We start with an overview of the design rationale (§4.1), followed by common patterns of finitizing the kernel interface (§4.2). The interface allows us to implement user-space libraries to support file systems and networking (§4.3). We end this section with a discussion of limitations (§4.4).

4.1 Design overview

To make the kernel interface finite, Hyperkernel combines OS design ideas from three main sources—Dune [8], exokernels [23, 33], and seL4 [34, 35]—as follows.

Processes through hardware virtualization. Unlike conventional Unix-like kernels, Hyperkernel provides the abstraction of a process using Intel VT-x and AMD-V virtualization support. The kernel runs as a host and user processes runs as guests (in ring 0), similarly to Dune [8]. Trap handlers are implemented as VM-exit handlers, in response to hypercalls (to implement system calls), preemption timer expiration, exceptions, and interrupts.

This approach has two advantages. First, it allows the kernel and user space to have separate page tables; the kernel simply uses an identity mapping for its own address space. Compared to previous address space designs (e.g., seL4 [34, 37]), this design sidesteps the need to reason about virtual-to-physical mapping for kernel code, simplifying verification.

Second, the use of virtualization safely exposes the interrupt descriptor table (IDT) to user processes. This allows the CPU to deliver exceptions (e.g., general protection or page fault) directly to user space, removing the kernel from most exception handling paths. In addition to performance benefits [8, 64], this design reduces the amount of kernel code that needs verification—Hyperkernel handles VM-exits due to "triple faults" from processes only, leaving the rest of exception handling to user space.

Explicit resource management. Similarly to exokernels [23], Hyperkernel requires user space to explicitly make resource allocation decisions. For instance, a system call for page allocation requires user space to provide a page number. The kernel simply checks whether the given resource is free, rather than searching for a free one itself.

This approach has two advantages. First, it avoids loops in the kernel and so makes verification scalable. Second, it can be implemented using array-based data structures, which the verifier can easily translate into SMT; it avoids reasoning about linked data structures (e.g., lists and trees), which are not well supported by solvers.

On the other hand, reclaiming resources often requires a loop, such as freeing all pages from a zombie process. To

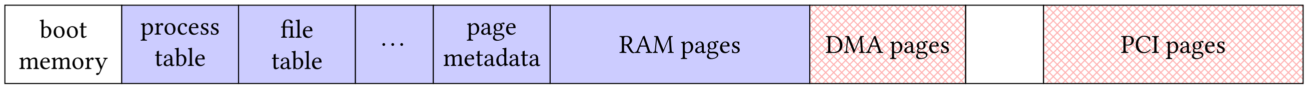

Figure 6: Memory layout in Hyperkernel: boot memory is used only during kernel initialization; shaded regions are accessible only by the CPU; and crosshatched regions are accessible by both the CPU and I/O devices.

keep the kernel interface finite, Hyperkernel safely pushes such loops to user space. For example, Hyperkernel provides a system call for user space to explicitly reclaim a page. In doing so, the kernel reclaims only a finite number of resources within each system call, avoiding long-running kernel operations altogether [22]. Note that the reclamation system call allows *any* process to reclaim a page from a zombie process, without the need for a special user process to perform garbage collection.

Typed pages. Figure 6 depicts the memory layout of Hyperkernel. It contains three categories of memory regions:

- Boot memory is used during kernel initialization only (e.g., for the kernel's identity-mapping page table) and freezes after booting.
- The main chunk of memory is used to keep kernel metadata for resources (e.g., processes, files, and pages), as well as "RAM pages" holding kernel and user data.
- There are two volatile memory regions: DMA pages, which restrict DMA (§3.1); and PCI pages (i.e., the "PCI hole"), which are mapped to device registers.

"RAM pages" are typed similarly to seL4: user processes retype pages through system calls, for instance, turning a free page into a page-table page, a page frame, or a stack. The kernel uses page metadata to track the type and ownership of each page and decide whether to allow such system calls.

4.2 Designing finite interfaces

We have designed Hyperkernel's system call interface following the rationale in §4.1. In particular, using xv6 and POSIX as our basis, we have made the Hyperkernel interface finite and amenable to push-button verification. This design leads to several common patterns described below. We also show how these patterns help ensure desired crosscutting properties, verified as part of the declarative specification.

Enforcing resource lifetime through reference counters. As mentioned earlier, the kernel provides system calls for user space to explicitly reclaim resources, such as processes, file descriptors, and pages. To avoid resource leaks, the kernel needs to carefully enforce their lifetime. For example, before it reaps a zombie process and reclaims its PID, the kernel needs to ensure that all the open file descriptors and pages associated with this process have been reclaimed, and that all its child processes have been re-parented (e.g., to `init`).

To do so, Hyperkernel reuses much of xv6's process structure and augments it with a set of reference counters to track its resource usage, such as the number of FDs, pages, and children. Reaping a process succeeds only if all of its reference counters are already zero.

As an example, recall that in §3.3 we have verified the correctness of the reference counter of the number of children:

Property 1. *For any process p, the value of its reference counter* `nr_children` *in the process structure must equal the number of processes with p as its parent.*

We use this property to ensure that user space must have re-parented all the children of a process before reaping the process, by proving the following:

Property 2. *If a process p is marked as free, no process designates p as its parent.*

We have verified similar properties for other resources that can be owned by a process. Such properties ensures that the kernel does not leak resources when it reaps a process.

Enforcing fine-grained protection. Some POSIX system calls have complex semantics. One example is `fork` and `exec` for process creation: `fork` needs to search for a free PID and free pages, duplicate resources (e.g., pages and file descriptors), and add the child process to the scheduler; `exec` needs to discard the current page table, read an ELF executable file from disk, and load it into memory. It is non-trivial to formally specify the behavior of these complex system calls let alone verify their implementations.

Instead, Hyperkernel provides a primitive system call for process creation in an exokernel fashion [33]. It creates a minimal process structure with three pages (i.e., the virtual machine control structure, the page table root, and the stack), leaving much of the work (e.g., resource duplication and ELF loading) to user-space libraries. This approach does not guarantee, for instance, that an ELF executable file is correctly loaded, though bugs in user-space libraries will be confined to that process.

This primitive system call is easier to specify, implement, and verify. As in exokernels, it still guarantees isolation. For example, we have proved the following in §3.3:

Property 3. *The page of the page table root of a process p is exclusively owned by p.*

Similarly, Hyperkernel exposes fine-grained virtual memory management. In particular, it provides page-table allocation through four primitive system calls: starting from the page table root, each system call retypes a free page and extends the page table to the next level. Memory allocation operations such as `mmap` and `brk` are implemented by user-space libraries using these system calls. This design is similar to seL4 and Barrelfish [7], but using page metadata rather than capabilities. The system calls follow xv6's semantics, where processes do not share pages. We have proved the following:

Property 4. *Each writable entry in a page-table page of process p must refer to a next-level page exclusively owned by p.*

Combining Properties 3 and 4, we have constructed an abstract model of page walking to prove the following memory isolation property:

Property 5. *Given any virtual address in process p, if the virtual address maps to a writable page through a four-level page walk, that page must be exclusively owned by p, and its type must be a page frame.*

This property ensures that a process can modify only its own pages, and that it cannot bypass isolation by directly modifying pages of critical types (e.g., page-table pages). We have proved similar properties for readable page-table entries and virtual addresses. In addition, Hyperkernel provides fine-grained system calls for managing IOMMU page tables, with similar isolation properties (omitted for brevity). §6.1 will describe bugs caught by these isolation properties.

Validating linked data structures. As mentioned earlier, Hyperkernel uses arrays to keep metadata. For instance, for page allocation the kernel checks whether a user-supplied page is free using an array for the page metadata; user space can implement its own data structures to find a free page efficiently. However, this technique does not preclude the use of linked data structures in the kernel. Currently, Hyperkernel maintains two linked lists: a free list of pages and a ready list of processes. They are not necessary for functionality, but can help simplify user-space implementations.

Take the free list as an example. The kernel embeds a linked list of free pages in the page metadata (mapped as read-only to user space). This free list serves as suggestions to user processes: they may choose to allocate free pages from this list or to implement their own bookkeeping mechanism. For the kernel, the correctness of page allocation is not affected by the use of the free list, as the kernel still validates a user-supplied page as before—if the page is not free, page allocation fails. This approach thus adds negligible work to the verifier, as it does not need to verify the full functional correctness of these lists.

4.3 User-space libraries

Bootstrapping. Like xv6, Hyperkernel loads and executes a special `init` process after booting. Unlike xv6, `init` has access to the IDT and sets up user-level exception handling. Another notable difference is that instead of using `syscall`, user space uses hypercall (e.g., `vmcall`) instructions to invoke the kernel. We have implemented a libc that is source compatible with xv6, which helps in porting xv6 user programs.

File system. We have ported the xv6 journaling file system to run on top of Hyperkernel as a dedicated file server process. The file system can be configured to run either as an in-memory file system or with an NVM Express disk, the driver for which uses IOMMU system calls provided by the kernel. While the file system is not verified, we hope to incorporate recent efforts in file system verification [3, 13, 62, 63] in the future.

Network. We have implemented a user-space driver for the E1000 network card (through IOMMU system calls) and ported lwIP [21] to run as a dedicated network server. We have implemented a simple HTTP server and client, capable of serving the git repository that hosts this paper.

Linux user emulation. We have implemented an emulator to execute unmodified, statically linked Linux binaries. Since the user space is running as ring 0, the emulator simply intercepts `syscall` instructions and mimics the behavior of Linux system calls, similar to Dune [8]. The current implementation is incomplete, though it can run programs such as gzip, sha1sum, Z3, and the benchmarks we use in §6.4.

4.4 Limitations

The use of hardware virtualization in Hyperkernel simplifies verification by separating kernel and user address spaces, and by pushing exception handling into user space. This design choice does not come for free: the (trusted) initialization code is substantially larger and more complex, and the overhead of hypercalls is higher compared to `syscall` instructions, as we evaluate in §6.4.

Hyperkernel's data structures are designed for efficient verification with SMT solvers. The kernel relies on arrays, since the verifier can translate them into uninterpreted functions for efficient reasoning. It uses linked data structures through validation, such as the free list of pages and the ready list of processes. Though this design is safe, the verifier does not guarantee that the free list contains all the free pages, or that the scheduler is free of starvation. Incorporating recent progress in automated verification of linked data structures may help with such properties [57, 71].

Hyperkernel requires a finite interface. Many POSIX system calls (e.g., `fork`, `exec`, and `mmap`) are non-finite, as they perform a number of operations. As another example, the seL4 interface, in particular revoking capabilities, is also non-finite, as it involves potentially unbounded loops over recursive data structures [22]. Verifying these interfaces requires more expressive logics and is difficult using SMT.

The Hyperkernel verifier works on LLVM IR. Its correctness guarantees therefore do not extend to the C code or the final binary. For example, the verifier will miss undefined behavior bugs in the C code if they do not manifest as undefined behavior in the LLVM IR (e.g., if the C frontend employs a specific interpretation of undefined behavior).

Finally, Hyperkernel inherits some limitations from xv6. It does not support threads, copy-on-write fork, shared pages, or Unix permissions. Unlike xv6, Hyperkernel runs on a uniprocessor system and does not provide multicore support. Exploring finite-interface designs to support these features is left to future work.

5 CHECKERS

The Hyperkernel theorems provide correctness guarantees for trap handlers. However, they do not cover kernel initialization or glue code. Verifying these components would require constructing a machine-level specification for x86 (including hardware virtualization), which is particularly challenging due to its complexity. We therefore resort to testing and analysis for these cases, as detailed next.

Boot checker. Theorem 1 guarantees that the representation invariant (e.g., `current` is always a valid PID) holds after each trap handler if it held beforehand. However, it does not guarantee that the invariant holds initially.

Recall that the verifier asks programmers to write the representation invariant in C in a special `check_rep_invariant` function (§2.2). To check that the representation invariant holds initially, we modify the kernel implementation to simply call this function before it starts the first user process `init`; the kernel panics if the check fails.

Similarly, Theorem 2 guarantees that the predicate P holds after each transition in the state-machine specification if it held beforehand. But it does not guarantee that P is not vacuous—P may never hold in any state. To preclude such a bug in the declarative specification, we show P is not vacuous by constructing an initial state and checking that it satisfies P.

Stack checker. LLVM IR does not model machine details such as the stack. Therefore, the verification of Hyperkernel, which is at the LLVM IR level, does not preclude run-time failures due to stack overflow. We implement a static checker for the x86 assembly generated by the LLVM backend; it builds a call graph and conservatively estimates the maximum stack space used by trap handlers. Running the checker shows that they all execute within the kernel stack size (4 KiB).

Link checker. The verifier assumes that symbols in the LLVM IR do not overlap. We implement a static checker to ensure that the linker maintains this property in the final kernel image. The checker reads the memory address and size of each symbol from the kernel image and checks that all the symbols reside in disjoint memory regions.

6 EXPERIENCE

This section reports on our experience in developing Hyperkernel, as well as its verification and run-time performance.

6.1 Bug discussion

To understand how effective the verifier and checkers are in preventing bugs in Hyperkernel, we examine the git commit log of xv6 from January 2016 to July 2017. Even though xv6 is a fairly mature and widely used teaching OS, during this time period the authors have found and fixed a total of ten bugs in the xv6 kernel. We manually analyze these bugs and determine whether they can occur in Hyperkernel. We exclude two lock-related bugs as they do not apply to Hyperkernel. Figure 7 shows the eight remaining bugs. In summary:

- Five bugs cannot occur in Hyperkernel: four would be caught by the verifier as they would trigger undefined behavior or violate functional correctness; and one buffer overflow bug would be caught either by the verifier or the boot checker.
- Three bugs are in xv6's `exec` system call for ELF loading; Hyperkernel implements ELF loading in user space, and thus similar bugs can happen there, but will not harm the kernel.

During the development of Hyperkernel, the verifier identified several bugs in our C code. Examples included incorrect assertions that could crash the kernel and missing sanity checks on system call arguments. The declarative specification also uncovered three interesting bugs in the state-machine specification, as described below.

The first bug was due to inconsistencies in the file table. To decide the availability of a slot in the file table, some system calls checked whether the reference count was zero (Figure 4), while others checked whether the file type was a special value (`FD_NONE`); in some cases both fields were not updated consistently. Both the state-machine specification and the implementation had the same bug. It was caught by the reference counting property (§2.2) in the declarative specification.

The other two bugs were caused by incorrect lifetime management in the IOMMU system calls. Hyperkernel exposes system calls to manage two IOMMU data structures: a device table that maps a device to the root of an IOMMU page table, and an IOMMU page table for address translation. To avoid

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.

Component	Lines	Languages
kernel implementation	7,419	C, assembly
representation invariant	197	C
state-machine specification	804	Python
declarative specification	263	Python
user-space implementation	10,025	C, assembly
verifier	2,878	C++, Python

Figure 8: Lines of code for each component.

dangling references, the kernel must invalidate a device-table entry *before* reclaiming pages of the IOMMU page table referenced by the entry. Initially, both the state-machine specification and the implementation failed to enforce this requirement, violating the isolation property (§4.2) in the declarative specification: an IOMMU page walk must end at a page frame (i.e., it cannot resolve to a free page). A similar bug was found in the system call for invalidating entries in the interrupt remapping table.

Our experience with the declarative specification suggests that it is particularly useful for exposing corner cases in the design of the kernel interface. We share some of Reid’s observations [60]: high-level declarative specifications capture properties across many parts of the system, which are often difficult for humans to keep track of. They are also more intuitive for expressing the design intent and easier to translate into natural language for understanding.

6.2 Development effort

Figure 8 shows the size of the Hyperkernel codebase. The development effort took several researchers about a year. We spent most of this time experimenting with different system designs and verification techniques, as detailed next.

System designs. The project went through three major revisions. We wrote the initial kernel implementation in the Rust programming language. Our hope was that the memory-safety guarantee provided by the language would simplify the verification of kernel code. We decided to abandon this plan for two reasons. One, Rust’s ownership model posed difficulties, because our kernel is single-threaded but has many entry points, each of which must establish ownership of any memory it uses [45]. Two, it was challenging to formalize the semantics of Rust due to the complexity of its type system—see recent efforts by Jung et al. [32] and Reed [59].

Our second implementation was in C, with a more traditional OS design on x86-64: the kernel resided at the top half of the virtual address space. As mentioned earlier, this design complicated reasoning about kernel code—every kernel pointer dereference needed to be mapped to the corresponding physical address. We started to look for architectural support that would simplify reasoning about virtual memory; ideally, it would allow us to run the kernel with an identity mapping, in a separate address space from user code.

Hyperkernel is our third attempt. We started with xv6, borrowing the idea of processes as guests from Dune [8], and tailoring it for verification. For instance, Dune uses the Extended Page Tables (EPT) and allows user space to directly control its own `%CR3`. Hyperkernel disallows the EPT due to its unnecessary complexity and address translation overhead, instead providing system calls for page-table management. With the ideas described in §4, we were able to finish the verification with a low proof burden.

Verification and debugging. As illustrated in §2, verifying a new system call in Hyperkernel (after implementing it in C) requires three steps: write a state-machine specification (in Python); relate data structures in LLVM IR to the abstract kernel state in the equivalence function (in Python); and add checks in the representation invariant if needed (in C).

Our initial development time was spent mostly on the first step. During that process, we developed high-level libraries for encoding common crosscutting properties (§3.3) and for mapping data structures in LLVM IR to abstract state. With these components in place, and assuming no changes to the declarative specification, one of the authors can verify a new system call implementation within an hour.

As it is unlikely to write bug-free code on the first try, we found that counterexamples produced by Z3 are useful for debugging during development. At first, the generated test case was often too big for manual inspection, as it contained the entire kernel state (e.g., the process table, the file table, as well as FDs and page tables). We then added support in the verifier for minimizing the state and highlighting offending predicates that were violated.

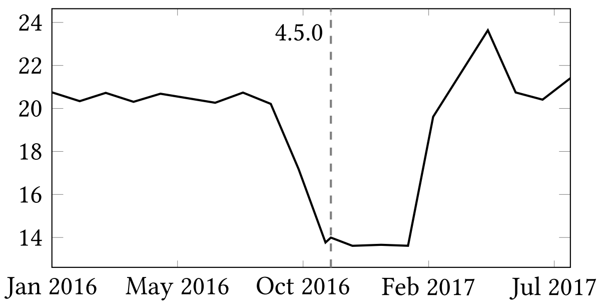

Figure 9: Verification time in minutes with Z3 commits over time. The dashed lines represent Z3 releases.

In our experience, Z3 was usually able to generate counterexamples for implementation bugs that violated Theorem 1. However, for violations of Theorem 2 (e.g., bugs in the state-machine specification), Z3 sometimes failed to do so if the size of the counterexample was too big. We worked around this issue by temporarily lowering system parameters for debugging (e.g., limiting the maximum number of processes or pages to a small value). This reduction helps Z3 construct small counterexamples, and also makes it easier to understand the bug. As hypothesized by Jackson [30], we found these "small counterexamples" sufficient for finding and fixing bugs in Hyperkernel.

SMT encodings. Hyperkernel benefits from the use of SMT solvers for verification. However, SMT is no silver bullet: naïve encodings can easily overwhelm the solver, and given a non-finite interface, there may not be efficient SMT encodings. At present, there is no general recipe for developing interfaces or encodings that are amenable to effective SMT solving. We hope that more work in this direction can help future projects achieve scalable and stable verification.

Based on lessons learned from the Yggdrasil file system verification project [63], we designed the kernel to be event driven with fine-grained event handlers (i.e., trap handlers). This design simplifies verification as it avoids reasoning about interleaved execution of handlers; it also limits the size of SMT expressions generated from symbolic execution. In addition, Hyperkernel restricts the use of quantifiers to a few high-level properties in the declarative specification (§3.3). Compared to other SMT-based verification projects like Ironclad [28] and IronFleet [27], this practice favors proof automation and verification scalability. The trade-off is that it requires more care in interface design and encodings, and it limits the types of properties that can be verified (§4.4).

6.3 Verification performance

Using Z3 4.5.0, verifying the two main Hyperkernel theorems takes about 15 minutes on an eight-core Intel Core i7-7700K processor. On a single core it takes a total of roughly 45 minutes: 12 and 33 minutes for verifying Theorems 1 and 2, respectively.

To understand the stability of this result, we also verified Hyperkernel using the first Z3 git commit in each month since January 2016. None of these versions found a counterexample to correctness. Figure 9 shows the verification time for these commits; the performance is mostly stable, with occasional spikes due to Z3 regressions and heuristic changes.

The Hyperkernel verification uses fixed values of some important constants (e.g., the maximum number of pages), derived from xv6. To check the robustness of the verification to changes in these parameters, we tried verifying Hyperkernel with different values to ensure that they did not cause substantial increases in verification time. In particular, we increased the maximum number of pages (the largest value among the parameters) by 2×, 4×, and 100×; and did not observe a noticeable increase in verification time. This result suggests that the finite-interface design described in §2.1 and the SMT encodings described in §3.3 are effective, as verification performance does not depend on the size of kernel state.

6.4 Run-time performance

To evaluate the run-time performance of Hyperkernel, we adopt benchmarks from Dune [8], excluding those that do not apply to Hyperkernel (e.g., ptrace). Figure 10 compares Hyperkernel run-time performance (in cycles) to Linux (4.8.0) on four benchmarks. The Hyp-Linux results run the unmodified Linux benchmark binary on Hyperkernel using the Linux emulation layer (§4.3), while the Hyperkernel results are from a ported version of the benchmark. All results are from an Intel Core i7-7700K processor.

The syscall benchmark measures a simple system call—`sys_nop` on Hyperkernel, and `gettid` (which performs minimal work) on Linux and Hyp-Linux. The 5× overhead on Hyperkernel is due to the overhead of making a hypercall, as measured in §6.5. Hyp-Linux's emulation layer services system calls within the same process, rather than with a hypercall, and so its performance is close to Linux.

The fault benchmark measures the cycles to invoke a user-space page fault handler after a fault. Hyperkernel outperforms Linux because faults can be delivered directly to user space, thanks to virtualization support; the Linux kernel must first catch the fault and then upcall to user space.

The appel1 and appel2 benchmarks, described by Appel and Li [5], measure memory management performance with repeated (un)protection and faulting accesses to protected pages. Hyperkernel outperforms Linux here for the same reason as the earlier fault benchmark.

These results are consistent with the comparison between Dune and Linux [8]: for the worst-case scenarios, the use

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).

of hypercalls incurs a 5× overhead compared to `syscall`; on the other hand, depending on the workload, application performance may also benefit from virtualization (e.g., fast user-level exceptions).

The current user-space file system and network stack (§4.3) is a placeholder for demonstrating the usability of the kernel interface. We hope to incorporate high-performance stacks [9, 56] in the future.

6.5 Reflections on hardware support

Hyperkernel's use of virtualization simplifies verification, but replaces system calls with hypercalls. To understand the hardware trend of `syscall` and hypercall instructions, we measured the round-trip latency on recent x86 microarchitectures. The results are shown in Figure 11: the "syscall" column measures the cost of a `syscall`/`sysret` pair, and the "hypercall" column measures the cost of a `vmcall`/`vmresume` pair (or `vmmcall`/`vmrun` on AMD).

Our observation is that while hypercalls on x86 are slower by approximately an order of magnitude due to the switch between root and non-root modes, their performance has significantly improved over recent years [1]. For non-x86 architectures like ARM, the system call and hypercall instructions have similar performance [18], and so exploring Hyperkernel on ARM would be attractive future work [31].

7 RELATED WORK

Verified OS kernels. OS kernel verification has long been a research objective. Early efforts in this direction include UCLA Secure Unix [66], PSOS [24], and KIT [10]; see Klein et al. [34] for an overview.

The seL4 verified kernel demonstrated, for the first time, the feasibility of constructing a machine-checkable formal proof of functional correctness for a general-purpose kernel [34, 35]. Hyperkernel's design is inspired in several places by seL4, as discussed in §4.1. In contrast to seL4, however, we aimed to keep Hyperkernel's design as close to a classic Unix-like kernel as possible, while enabling automated verification with SMT solvers. In support of this goal, we made the Hyperkernel interface finite, avoiding unbounded loops, recursion, or complex data structures.

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.

Ironclad establishes end-to-end security properties from the application layer down to kernel assembly [28]. Ironclad builds on the Verve type-safe kernel [69], and uses the Dafny verifier [44], which is built on Z3, to help automate proofs. Similarly, ExpressOS [51] verifies security properties of a kernel using Dafny. As discussed in §6.2, Hyperkernel focuses on system designs for minimizing verification efforts and restricts its use of Z3 for better proof automation. Hyperkernel also verifies at the LLVM IR layer (trusting the LLVM backend as a result) rather than Verve's assembly.

Examples of recent progress in verifying *concurrent* OS kernels include CertiKOS with multicore support [25] and Xu et al.'s framework for reasoning about interrupts in the μC/OS-II kernel [68]. Both projects use the Coq interactive theorem prover [16] to construct proofs, taking 2 and 5.5 person-years, respectively. The Hyperkernel verifier does not reason about multicore or interrupts in the kernel. Investigating automated reasoning for concurrent kernels would be a promising direction.

Co-designing systems with proof automation. As discussed in §6.2, Hyperkernel builds on the lessons learned from Yggdrasil [63], a toolkit for writing file systems and verifying them with the Z3 SMT solver. Yggdrasil defines file system correctness as crash refinement: possible disk states after a crash are a subset of those allowed by the specification. In contrast, Hyperkernel need not model crashes, but needs to reason about reference counting (§3.3). Moreover, Yggdrasil file systems are implemented and verified in Python; Hyperkernel is written in C and verified as LLVM IR, removing the dependency on the Python runtime from the final binary.

Reflex automates the verification of event-driven systems in Coq [61], by carefully restricting both the expressiveness of the implementation language and the class of properties to be verified. Hyperkernel and Reflex share some design principles, such as avoiding unbounded loops. But thanks to its use of SMT solvers for verification, Hyperkernel can prove a richer class of properties, at the cost of an enlarged TCB.

OS design. Hyperkernel borrows ideas from Dune [8], in which each process is virtualized and so has more direct control over hardware features such as interrupt vectors [8]. This allows Hyperkernel to use a separate identity mapping for the kernel address space, avoiding the need to reason about virtual-to-physical mapping. Hyperkernel also draws inspiration from Exokernel [23], which offers low-level hardware access to applications for extensibility. As discussed in §4.1, this design enables finite interfaces and thus efficient SMT-based verification.

LLVM verification. Several projects have developed formal semantics of LLVM IR. For example, the Vellvm project [72] formalizes LLVM IR in Coq; Alive [49] formalizes LLVM IR to verify the correctness of compiler optimizations; SMACK [58] translates LLVM IR to the Boogie verification language [6]; KLEE [11] and UFO [2] translate LLVM IR to SMT for verification; and Vigor uses a modified KLEE as part of its toolchain to verify a network address translator [70].

The Hyperkernel verifier also translates LLVM IR to SMT to prove Theorem 1 (§3.2). Compared to other approaches, Hyperkernel's verifier unrolls all loops and recursion to simplify automated reasoning, requiring programmers to ensure kernel code is self-finitizing [65]. The verifier also uses a simple memory model tailored for kernel verification, translating memory accesses to uninterpreted functions.

8 CONCLUSION

Hyperkernel is an OS kernel formally verified with a high degree of proof automation and low proof burden. It achieves push-button verification by finitizing kernel interfaces, using hardware virtualization to simplify reasoning about virtual memory, and working at the LLVM IR level to avoid modeling C semantics. Our experience shows that Hyperkernel can prevent a large class of bugs, including those previously found in the xv6 kernel. We believe that Hyperkernel offers a promising direction for future design of verified kernels and other low-level software, by co-designing the system with proof automation. All of Hyperkernel's source code is publicly available at http://locore.cs.washington.edu/hyperkernel/.

ACKNOWLEDGMENTS

We thank Tom Anderson, Anish Athalye, Jon Howell, the anonymous reviewers, and our shepherd, Gernot Heiser, for their feedback. We also thank Alex Kirchhoff for the initial implementation of IOMMU support, John Regehr for discussions on undefined behavior in LLVM IR, and Marc Zyngier for answering our questions about ARM. This work was supported in part by DARPA under contract FA8750-16-2-0032.

REFERENCES

[1] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. 2012. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, 373–385.

[2] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. Berkeley, CA, 672–678.

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

[4] AMD. 2017. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Rev. 3.28.

[5] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Santa Clara, CA, 96–107.

[6] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. Lisbon, Portugal, 82–87.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, 29–44.

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

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 49–65.

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

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

[12] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*. Shanghai, China. 5 pages.

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

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. Chateau Lake Louise, Banff, Canada, 73–88.

[15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, 1–17.

[16] Coq development team. 2017. *The Coq Proof Assistant Reference Manual, Version 8.6.1*. INRIA. http://coq.inria.fr/distrib/current/refman/.

[17] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. 2016. Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/xv6.

[18] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, 333–347.

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

[20] Leonardo de Moura and Nikolaj Bjørner. 2011. Z3 - a Tutorial.

[21] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science.

[22] Kevin Elphinstone and Gernot Heiser. 2013. 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)*. Farmington, PA, 133–150.

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

[24] R. J. Feiertag, K. N. Levitt, and L. Robinson. 1977. Proving multilevel security of a system design. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles (SOSP)*. West Lafayette, IN, 57–65.

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

[26] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, 336–345.

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

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

[29] Intel. 2016. *Intel Virtualization Technology for Directed I/O: Architecture Specification*. Rev. 2.4.

[30] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

[31] Dylan Johnson. 2017. *Porting Hyperkernel to the ARM Architecture*. Technical Report UW-CSE-17-08-02. University of Washington.

[32] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 6th ACM SIGPLAN Workshop on Higher-Order Programming*. Oxford, United Kingdom.

[33] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France, 52–65.

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

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

[36] Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 323–339.

[37] Rafal Kolanski. 2011. *Verification of Programs in Virtual Memory Using Separation Logic*. Ph.D. Dissertation. University of New South Wales.

[38] Rafal Kolanski and Gerwin Klein. 2009. Types, Maps and Separation Logic. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Munich, Germany, 276–292.

[39] Robbert Krebbers. 2015. *The C standard formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.

[40] Robbert Krebbers and Freek Wiedijk. 2012. *Subtleties of the ANSI/ISO C standard*. Document N1637. ISO/IEC JTC1/SC22/WG14.

[41] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

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

[43] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, 633–647.

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

[45] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is Theft: Experiences Building an Embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. Monterey, CA, 21–26.

[46] Henry M. Levy and Peter H. Lipman. 1982. Virtual Memory Management in the VAX/VMS Operating System. *Computer* 15, 3 (March 1982), 35–41.

[47] Linux Programmer's Manual 2016. `dup`, `dup2`, `dup3` - duplicate a file descriptor. http://man7.org/linux/man-pages/man2/dup.2.html.

[48] John Lions. 1996. *Lions' Commentary on Unix* (6th ed.). Peer-to-Peer Communications.

[49] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, 22–32.

[50] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A Study of Linux File System Evolution. *ACM Transactions on Storage* 10, 1 (Jan. 2014), 31–44.

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

[52] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2014. System V Application Binary Interface: AMD64 Architecture Processor Supplement, Draft Version 0.99.7.

[53] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, 1–15.

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

[55] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 305–318.

[56] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 1–16.

[57] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and P. Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, 16–19.

[58] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*. Vienna, Austria, 106–113.

[59] Eric Reed. 2015. *Patina: A Formalization of the Rust Programming Language*. Technical Report UW-CSE-15-03-02. University of Washington.

[60] Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the ARM v8-M Architecture Specification. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Vancouver, Canada, 88:1–24.

[61] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. Automating Formal Proofs for Reactive Systems. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, UK, 452–462.

[62] Gerhard Schellhorn, Gidon Ernst, Jorg Pfähler, Dominik Haneberg, and Wolfgang Reif. 2014. Development of a Verified Flash File System. In *Proceedings of the ABZ Conference*. Toulouse, France, 9–24.

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

[64] Chandramohan A. Thekkath and Henry M. Levy. 1994. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, 110–119.

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

[66] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. 1980. Specification and Verification of the UCLA Unix Security Kernel. *Commun. ACM* 23, 2 (Feb. 1980), 118–131.

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

[68] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*. Toronto, Canada, 59–79.

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

[70] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the 2017 ACM SIGCOMM Conference*. Los Angeles, CA, 141–154.

[71] Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2008. Full Functional Verification of Linked Data Structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Tucson, AZ, 349–361.

[72] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)*. Philadelphia, PA, 427–440.