PUSH-BUTTON VERIFICATION OF SYSTEMS SOFTWARE

HELGI KRISTVIN SIGURBJARNARSON

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

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Xi Wang, Chair

Emina Torlak

Henry M. Levy

Program Authorized to Offer Degree:

Paul G. Allen School of Computer Science & Engineering

University of Washington

ABSTRACT

PUSH-BUTTON VERIFICATION OF SYSTEMS SOFTWARE

Helgi Kristvin Sigurbjarnarson

Chair of the Supervisory Committee:
Associate Professor Xi Wang
Paul G. Allen School of Computer Science & Engineering

Systems software interfaces with hardware, multiplexes resources, and provides common abstractions for modern applications to build on. The correctness and reliability of these systems are critical for the applications and users that depend on them. While formal verification of systems software can be effective at eliminating bugs, it is a non-trivial task, requiring developers to write many lines of proof code for every line of implementation code. This is a considerable engineering effort that requires a high degree of expertise to achieve.

This dissertation explores a new approach to designing, specifying, implementing, and verifying systems software in a *push-button* fashion. By co-designing systems software with automation, we argue it is possible to build correct and reliable systems with substantially less effort. We developed four systems to demonstrate the effectiveness of this approach, using the Z3 satisfiability modulo theories (SMT) solver. First, we developed Yggdrasil, a toolkit for writing file systems. Yggdrasil uses push-button verification with a new definition of file system correctness called *crash refinement*. Crash refinement is amenable to fully automated reasoning, and it enables developers to implement file systems in a modular way for verification. Second, we developed an OS kernel named Hyperkernel, which has a high degree of proof automation and low proof burden. 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. Third, we developed Nickel, a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface. Nickel provides a formulation of noninterference amenable to automated verification, allowing developers to specify an intended policy of permitted information flows. Fourth, we present Ratatoskr, showing the feasibility of applying push-button verification to distributed protocols. Together, these contributions demonstrate the effectiveness of treating automation as a first-class design principle, letting developers build verified systems software with substantially less effort.

Til Helgu

ACKNOWLEDGEMENTS

I am tremendously fortunate to have had the opportunity to collaborate with and learn from so many wonderful people. First, I would like to thank my advisor, Xi Wang, to whom I'm immensely grateful. From our first introduction and throughout my Ph.D., Xi has been incredibly generous with his time and happy to entertain my ideas and chat about anything and everything–even when that meant taking my call at 1 am on a Saturday to hash out the gnarly details of information flow meta-theory. His deep technical insight and always concrete, constructive feedback, advice, encouragement, and guidance have had an immeasurable impact on my work and my life, imparting me with an invaluable skill set that I take with me into the future.

From my first quarter at UW, I have had the privilege to collaborate with Emina Torlak, who has been like a second advisor to me. Her boundless knowledge of all things solver-aided and formal methods was essential to making any of this work. Whenever things didn't pan out as I had expected, I could always run to Emina for advice on how to placate the arcane SMT deities. I also want to thank the other members of my committee: Scott Hauck and Hank Levy. Hank, from my first visit, always made me feel welcome in the Allen school, providing a friendly face along with helpful feedback on my work and presentation skills.

Additionally, the work in this thesis is the result of a collaboration with my friends James Bornholt and Luke Nelson, with countless brainstorms, hours spent hacking on prototypes and running experiments. This work was also enhanced by feedback from multiple other people along the way, and I'm especially appreciative of guidance from Jon Howell and Jay Lorch.

I want to thank all the amazing students and faculty that I got to know, learn from, and work with at the Allen School through the years, especially: Ellis Michael, Adriana Szekeres, Anna Kornfeld Simpson, Antoine Kaufmann, Arvind Krishnamurthy, Ashlie Martinez, Bruno Castro-Karney, Dan Ports, Danyang Zhuo, Dylan Johnson, Irene Zhang, Jacob Van Geffen, Jared Roesch, Jialin Li, Kaiyuan Zhang, Kevin Zhao, Maaz Ahmad, Naveen Kr. Sharma, Niel Lebeck, Pedro Fonseca, Qiao Zhang, Ras Bodik, Samantha Miller, Simon Peter, Sorawee Porncharoenwase, Tom Anderson, Yuchen Jin, and Zachary Tatlock. All of these people have enriched my time here in so many ways. I also want to thank the administrative problem-solvers that made my life easier so many times, especially Elise Dorough and Melody Kadenko.

I will not attempt to enumerate all of the different things, people, and happenstance, that led me to where I am today. However, without two professors that I worked with during my undergraduate studies, my life could have taken a completely different course. Henning Ulfarsson was the first one to pique my interest in pursuing research, and even introduced me to Coq. Ymir Vigfusson, as my undergraduate advisor, took that interest and pushed me out of my comfort zone to pursue it in a much bigger way than I had imagined. Without him, I can say for certain that I would not have ended up in the Allen school in the first place.

Finally, Helga Guðmundsdóttir, my best friend and soulmate. I am thankful for your endless patience (even as you listened to my practice talks for the nth time) and being my sounding board. Thank you for keeping me motivated and being my source of inspiration, your great ideas, for putting up with my shenanigans, and for everything else you have done for me. But I am especially grateful for your love. Thanks for all the adventure–without you, I would be truly lost.

PUBLICATIONS

Portions of this thesis appeared in the following publications:

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

[2] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. "Hyperkernel: Push-Button Verification of an OS Kernel." In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, Oct. 2017, pp. 252–269.

[3] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. "Nickel: A Framework for Design and Verification of Information Flow Control Systems." In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, Oct. 2018, pp. 287–306.

CONTENTS

1 INTRODUCTION

Systems software provides the base layer of abstraction for user applications: a critical foundation for modern software. For example, applications depend on the operating system to both safely mediate access to hardware and correctly multiplex resources among mutually distrustful parties. Similarly, applications depend on the file system to persist data in the face of non-deterministic crashes or power failures. Consequently, bugs in systems software can have catastrophic consequences, ranging from unexpected behavior in applications to serious data losses or even allowing a malicious actor to compromise the entire system [22, 25, 29, 30, 100, 120, 124, 136, 160, 162, 173].

Formal verification provides a systematic approach to building correct and reliable systems. Previous research has applied it to building system software and shown it to be a powerful tool [5, 23, 55, 61, 62, 78, 79, 141, 159]. By constructing a trusted specification [79, 159] that describes the intended behavior of a system along with a machine-checkable proof that the implementation behaves according to that specification, entire classes of bugs can be ruled out. Commonly, the specification is much simpler than the implementation and written in a high-level language without low-level implementation details. As such, it is easier to read and audit for correctness. Assuming the specification is free of bugs, this verification approach assures that the verified system is correct with a high degree of confidence.

Applying formal verification, however, comes with a non-trivial cost. A common approach to verifying systems software is to ask the developer to provide a proof of correctness using an interactive theorem prover such as Coq [149] or Isabelle [118]. This often presents an enormous engineering challenge, requiring developers to spend a considerable amount of time writing proofs and annotations. For example, the functional correctness proof of the verified seL4 kernel took roughly 11 person-years to verify 10,000 lines of C code [78].

Another approach to building verified systems software uses auto-active verifiers [94] such as Dafny [93]. These require developers to write proof annotations that include pre- and postconditions, loop invariants, and frame annotations to guide the verifier. Although auto-active verifiers can reduce the proof burden compared to interactive theorem provers, they still require a substantial effort. Hawblitzel et al. reported writing between seven and eight lines of proof code for every line of code for the IronFleet implementation of Paxos [62] and achieved a proof-to-implementation ratio of 4.8:1 in the verified Ironclad Apps [61].

This dissertation explores a push-button approach to designing, specifying, implementing, and verifying systems: our goal is to substantially reduce the manual proof burden required to verify systems software. Not all system designs lend themselves equally to verification in terms of verification complexity, techniques, or tools necessary to certify correctness. In contrast to interactive or auto-active verification, push-button methodology shifts the burden of manual proofs and annotations to the design phase, which has to satisfy three key requirements. First, to make automated reasoning possible, the specification must be expressible in first-order logic. Second, the system must have a *finite interface*: an interface that can be implemented without the use of unbounded loops or recursion. These two requirements enable the use of symbolic execution to automatically translate the implementation into constraints, which are passed to a solver to check against the specification. Even then, a large and complicated set of constraints is likely to overwhelm the solver, leading to verification timeouts. So finally, to scale verification to complete systems, it is important to decompose the implementation into smaller units for modular verification.

By treating proof automation as a first-class design goal during each step of interface design, specification, and implementation, we can trade off generality in favor of a high degree of automation. A key insight at the core of this thesis is that the co-design of systems with automated verification in this way makes it *both possible and practical to design, specify, implement, and verify systems software in a push-button fashion.*

To demonstrate this, we developed techniques to scale verification to complete systems–including file systems, operating system kernels, and information flow control systems–using the state-of-the-art satisfiability modulo theories (SMT) solver, Z3 [109].

1.1 YGGDRASIL

Chapter 3 presents Yggdrasil, a toolkit that helps developers write file systems and formally verify their correctness in a *push-button* fashion. Yggdrasil asks developers for three inputs: a specification of the expected behavior, an implementation, and consistency invariants that indicate whether a file system image is in a consistent state. It then performs verification to check if the implementation meets the specification. If there is a bug, Yggdrasil produces a counterexample to help identify and fix the cause. If the verification passes, Yggdrasil produces an executable file system. It requires *no* manual annotations or proofs about the implementation code.

Yggdrasil scales up automated reasoning for verifying file systems with the idea of *crash refinement*, a new definition of file system correctness. Crash refinement captures the notion that even in the presence of non-determinism, such as system crashes and reordering of writes, any disk state produced by a correct implementation must also be producible by the specification (see Section 3.2 for a formal definition). Yggdrasil formulates file system verification as an SMT problem and invokes a state-of-the-art SMT solver to fully automate the proof process.

We used Yggdrasil to implement and verify Yxv6+sync, a journaling file system that resembles xv6 [32] and FSCQ [23], and Yxv6+group_commit, an optimized variant with relaxed crash consistency [18, 122].

1.2 HYPERKERNEL

Chapter 4 describes Hyperkernel, an approach to designing, implementing, and formally verifying the functional correctness of an OS kernel 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.

A key challenge in verifying Hyperkernel is 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 be implementable in a way that enables fully automated verification of such properties with an SMT solver.

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 verified the implementation of Hyperkernel with the Z3 SMT solver, checking 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.

1.3 NICKEL

Chapter 5 describes Nickel, a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface. These channels can be exploited to circumvent the enforcement of information flow policies. Nickel's formulation of noninterference is amenable to automated verification, allowing developers to specify an intended policy of permitted information flows. It invokes the Z3 SMT solver to verify that both an interface specification and an implementation satisfy noninterference with respect to the policy; if verification fails, it generates counterexamples to illustrate covert channels that cause the violation.

Using Nickel, we have designed, implemented, and verified NiStar, the first OS kernel for decentralized information flow control that provides: (1) a precise specification for its interface, (2) a formal proof that the interface specification is free of covert channels, and (3) a formal proof that the implementation preserves noninterference. We also applied Nickel to verify isolation in a small OS kernel, NiKOS, and reproduce known covert channels in the ARINC 653 avionics standard. Our experience shows that Nickel effectively identified and ruled out covert channels and successfully verified noninterference for systems with a low proof burden.

1.4 RATATOSKR

Chapter 6 presents Ratatoskr. Ratatoskr is a protocol specification and proof of correctness for the Disk Paxos consensus algorithm [48]. We have verified that the Ratatoskr protocol specification correctly achieves consensus, using the Z3 automated theorem prover. The verification of Ratatoskr identified a bug in the algorithm as described by Gafni and Lamport [48].

1.5 OUTLINE AND CONTRIBUTIONS

The main contributions of this thesis are the push-button verification methodology that enables a high degree of proof automation for systems software verification, four case studies applying the methodology, and a discussion of its limitations.

The rest of this dissertation is organized in the following chapters. Chapter 2 introduces the tools and techniques used throughout this dissertation, including the Z3 [109] automated theorem prover. Chapter 3 presents Yggdrasil, a toolkit for writing file systems with push-button verification. Chapter 4 describes Hyperkernel, a verified OS kernel using push-button verification. Chapter 5 presents Nickel, a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface. In Chapter 6 we discuss future work and preliminary results when applying push-button techniques to a replicated log, while Chapter 7 reflects on different design decisions, approaches and insights we gained. Finally, Chapter 8 presents concluding remarks.

2 BACKGROUND

This chapter provides a birds-eye view of the general ideas, concepts, and background material to formal verification of systems. We describe a common approach to functional verification, how we model systems for verification, and compare and contrast interactive and auto-active verification to the push-button methodology.

Systems, such as OS kernels and file systems, are commonly modeled as state machines consisting of a state (memory, registers, devices, etc.) and a set of transition handlers. Functionally, we can think of the transition handlers as pure functions, taking the current state of the system and the event as input, producing the new state as output. For an OS kernel, transition handlers include user operations (reading and writing memory), traps, faults, and interrupts.

To prove an implementation correct, we need a ground-truth definition of correctness, a *specification*. Finding the right specification is a hard problem in general. One way to prove functional correctness of systems relies on establishing an equivalence between an abstract high-level description of the system and the low-level implementation. Proving such an equivalence is known as *refinement* and is widely used for verifying the functional correctness of systems. Verification does not mean that the implementation is free of bugs; in the abstract, it merely means that the specification and the implementation are equally buggy. In practice, however, it rules out a large class of bugs. For one, developers generally write the specification in a well defined, high-level language ruling out low-level bugs which are not necessarily ruled out by the implementation language, such as undefined behavior, buffer overflows, and null pointer dereferences. Second, the specification is often much smaller and more abstract, making it easier to reason about and audit for correctness. Third, optional safety invariants such as pre- and postconditions established on the high-level specification are preserved through refinement to the implementation ruling out more classes of bugs and further increasing confidence in its correctness.

Next, we provide an overview of interactive, auto-active, and push-button verification.

2.1 INTERACTIVE VERIFICATION

Interactive theorem provers, such as Coq [149] and Isabelle [118], provide developers with powerful and expressive higher-order logics to make mathematical statements about programs. Verification conditions are generated and presented to the developer in an interactive session. Based on foundational reasoning, developers then manually construct proofs by composing previously proven theorems and lemmas–starting from a small set of pre-defined axioms–which are machine-checked by the theorem prover for correctness. In principle, this provides a high assurance of correctness, only relying on a small set of trusted components: a small checker at the core of the theorem prover. Writing such proofs, however, requires both a high degree of expertise and substantial time investment.

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 OS kernel [78, 79]. The functional correctness proof took roughly 11 person-years for 10,000 lines of C code, requiring about 200,000 lines of proofs using the Isabelle/HOL theorem prover. The functional correctness of seL4 consists of three layers: the abstract specification, the executable specification, and the implementation. The abstract specification is written in Isabelle and uses only high-level data structures providing flexibility to the implementation by leaving much low-level implementation detail unspecified. The seL4 implementation uses a subset of the C language. The semantics of the subset are defined in Isabelle and the implementation is parsed into the theorem prover for reasoning. The seL4 executable specification refines the abstract specification and the C implementation refines the executable specification. Consequently, since refinement is transitive, the C implementation refines the abstract specification.

2.2 AUTO-ACTIVE VERIFICATION

Auto-active verifiers [94], such as Dafny [93], ask developers to provide Hoar-style annotations on the implementation code. These annotations come in the form of loop invariants, pre- and postconditions, and frame annotations describing what part of the state a function may read or write. The verifier compiles the program and annotations, generating verification conditions and checking them with an automated theorem prover such as Z3 [109]. This use of a solver reduces the manual proof burden compared to interactive theorem proving, but requiring developers to come up with these annotations still comes with a substantial cost.

The Ironclad [61] project is a full-stack verification; verifying applications, libraries, and drivers, down to the kernel level, with proofs covering the final assembly code. The total verification effort took 3 person-years, requiring about 33,000 lines of proof annotations for about 7,000 lines of implementation code. The Ironclad implementation, specification, and annotations are written in Dafny, and compiled to BoogieX86 [159], a verifiable assembly language. The Boogie verifier then checks the assembly implementation against its specification. In addition to functional correctness, they additionally proved key security theorems, including noninterference.

2.3 PUSH-BUTTON VERIFICATION

This dissertation proposes a different approach to verifying systems software. Instead of asking developers to provide proofs or annotations to guide a verifier, push-button verification instead treats full automation for verification as a first-class design goal. Similar to auto-active verification, push-button verification leverages an automated SMT solver such as Z3 for proofs. However, instead of requiring developers to provide annotations, push-button verification aims for full automation in favor of generality or expressivity.

SMT solvers take as input a set of constraints and produce as output an assignment (if one exists) to the variables such that the constraints are satisfied. Program verification essentially boils down to showing that a program P behaves according to a specification `safe` on all possible inputs. That is, for a specification `safe` and program P, the goal of verification is to show that $\forall x.\ \texttt{safe}(x, P(x))$. Equivalently, we can view verification as a constraint problem, asking if there exists an input where the program violates the specification. As such, we can ask a solver to find a satisfying assignment to the formula $\exists x.\ \neg\texttt{safe}(x, P(x))$. If a solution exists, we refer to it as a *counterexample*, describing the exact input for which the program misbehaves. If no solution exists, we can conclude that the program is safe on all inputs, satisfying the specification. Push-button verification leverages this idea but require us to satisfy three conditions, described next. Compared to interactive and auto-active verification, these conditions trade off generality and expressivity for full automation.

EFFECTIVELY DECIDABLE FIRST-ORDER LOGIC Automated theorem provers are fundamentally limited to less expressive first-order logic. Leveraging an SMT solver in this fashion, therefore, requires the specification (`safe` in the above example) to be expressible in first-order logic. While modern solvers continue to gain features, expanding their expressiveness with complex logic and arbitrary explicit quantifiers, in practice, they will not be able to solve all queries using those features. To gain good verification performance, we limit our queries to only using an effectively decidable fragment of first-order logic: bitvectors, booleans, uninterpreted functions, and use quantifiers sparingly (over finite domains). This limitation may require us to rethink our specification if it is not directly expressible in this fragment. For instance, Nickel (Chapter 5) describes the meta-theory for an unwinding strategy for decomposing a complicated noninterference specification, making it amenable to automated verification.

FINITE INTERFACE Automatically producing a summary of the implementation $P(x)$ is feasible by using a technique known as symbolic execution. Symbolic execution is the process of compiling a program into a symbolic representation, describing how the program behaves under all possible inputs [76]. The verifiers in this thesis do this by implementing a symbolic interpreter for the target language (such as LLVM [91]). The symbolic interpreter operates on symbolic values instead of concrete ones, exhaustively exploring all execution paths. To perform an exhaustive execution means the implementation must be *finite*: free of unbounded loops and recursion. Hyperkernel (Chapter 4) describes the principles of *finite interface design* for achieving this goal.

DECOMPOSITION Symbolically executing a complicated implementation with many paths produces a large number of constraints that are likely to overwhelm the solver. To make verification feasible for a non-trivial system, we need a strategy for decomposing the implementation for modular verification. The first strategy is to break up the system into a collection of event handlers, bounding reasoning to a single handler at a time. This design lends itself well to push-button verification, provided each handler is sufficiently small for verification. This, dissertation describes two other strategies for decomposing *individual* handlers that may be too complicated for automated reasoning. Yggdrasil (Chapter 3) describes stacking layers of abstraction to scale verification, where the verification proceeds in layers, each layer refining the one before it, bounding the reasoning to a single layer at a time. In contrast, Hyperkernel (Chapter 4) describes decomposing complicated operations into smaller ones, exposing many operations, each of which does a relatively small amount of work, while still satisfying the top-level safety invariants.

PUTTING IT ALL TOGETHER To apply push-button verification to prove the functional correctness of an OS kernel, the developer writes both the specification and the implementation, as well as an equivalence relation $\approx$, relating the specification state s_{spec} and implementation state s_{impl}. An automated verifier performs symbolic execution, summarizing the specification and implementation as f_{spec} and f_{impl}, respectively. It then generates the refinement constraints by asking the solver if starting from equivalent states, executing the same operation, the specification and implementation can produce diverge results:

$$\exists s_{spec}, s_{impl}, x.\ s_{spec} \approx s_{impl} \wedge f_{spec}(s_{spec}, x) \not\approx f_{impl}(s_{impl}, x).$$

If a solution exists, the solver produces a counterexample that can be used for debugging. Otherwise, verification succeeds and we say that the implementation refines the specification.

Satisfying all of the requirements laid out above is challenging for any non-trivial system. The rest of this dissertation addresses those challenges, describing several techniques for push-button verification of systems software.

2.4 TCB

Verified software is *not* necessarily free of bugs. The specification may not accurately capture the intended properties or the verification tools, hardware, or any other unverified code (such as initialization or glue code) may be buggy. All parts of the system that are not explicitly verified and instead assumed to be correct are known as the trusted computing base (TCB). For all of the systems in this thesis, the TCB includes the Z3 solver, the Python interpreter, the symbolic execution engine, the compiler, and hardware it executes on.

2.5 CONCLUSION

Each verification tool and methodology has its own pros and cons. Common approaches and tools, such as interactive and auto-active verification, while general, come with a substantial development cost. By co-designing the system with automated verification and shifting the effort from manual proofs and annotations to interface design, specification and implementation, we can lower the overall cost substantially, while achieving high correctness assurance.

3 YGGDRASIL: PUSH-BUTTON VERIFICATION OF FILE SYSTEMS VIA CRASH REFINEMENT

The file system is an essential operating system component for persisting data on storage devices. Writing bug-free file systems is non-trivial, as they must correctly implement and maintain complex on-disk data structures even in the presence of system crashes and re-orderings of disk operations.

This chapter presents Yggdrasil, a toolkit for writing file systems with *push-button* verification: Yggdrasil requires no manual annotations or proofs about the implementation code, and it produces a counterexample if there is a bug. Yggdrasil achieves this automation through a novel definition of file system correctness called *crash refinement*, which requires the set of possible disk states produced by an implementation (including states produced by crashes) to be a subset of those allowed by the specification. Crash refinement is amenable to fully automated satisfiability modulo theories (SMT) reasoning, and enables developers to implement file systems in a modular way for verification.

With Yggdrasil, we have implemented and verified the Yxv6 journaling file system, the Ycp file copy utility, and the Ylog persistent log. Our experience shows that the ease of proof and counterexample-based debugging support make Yggdrasil practical for building reliable storage applications.

3.1 OVERVIEW

Figure 1 shows the Yggdrasil development flow. Programmers write the specification, implementation, and consistency invariants all in the same language (a subset of Python in our current prototype; see Section 3.2.2). If there is any bug in the implementation or consistency invariants, the verifier generates a counterexample to visualize it. For better run-time performance, Yggdrasil optionally performs optimizations (either built-in or written by developers) and re-verifies the code. Once the verification passes, Yggdrasil emits C code, which is then compiled and linked using a C compiler to produce an executable file system, as well as an `fsck` checker.

This section gives an overview of each of these steps, using a toy file system called YminLFS as a running example. We will show how to specify, implement, verify, and debug it; how to optimize its performance; and how to get a running file system mounted via FUSE [43].

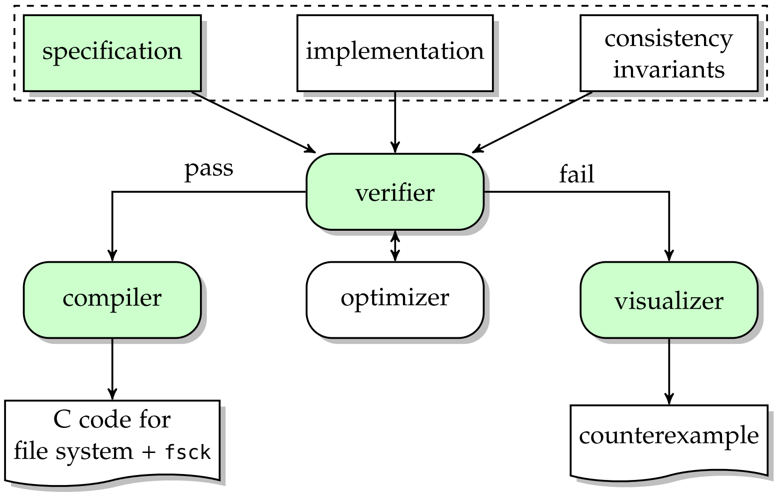


Figure 1: The Yggdrasil development flow. Rectangular boxes (within the dashed frame) denote input written by programmers; rounded boxes denote Yggdrasil's components; and curved boxes denote output. Shaded boxes are trusted to be correct and the rest are untrusted.

YminLFS is a log-structured file system [134]. It is kept minimal for demonstration purposes: there are no segments, subdirectories, or garbage collection, and files are zero-sized (no read, write, or unlink). But its core functionality is still tricky to implement correctly due to non-determinism and corner cases like overflows. In fact, the verifier caught two bugs in our initial implementation. The development of YminLFS took one of the authors less than four hours, as detailed next.

3.1.1 *Specification*

In Yggdrasil, a file system specification consists of three parts: an abstract data structure representing the logical layout, a set of operations over this data structure to define the intended behavior, and an equivalence predicate that defines whether a given implementation satisfies the specification.

ABSTRACT DATA STRUCTURE. We start by specifying the abstract data structure for YminLFS:

```
class FSSpec(BaseSpec):
  def __init__(self):
    self._childmap  = Map((InoT, NameT), InoT)
    self._parentmap = Map(InoT, InoT)
    self._mtimemap  = Map(InoT, U64T)
    self._modemap   = Map(InoT, U64T)
    self._sizemap   = Map(InoT, U64T)
```

The state of the data structure is described by five *abstract maps*, created by calling the `Map` constructor with *abstract types* specifying the map's domain and range. The `childmap` maps a directory inode number and a name to a child inode number; `parentmap` maps an inode number back to its parent directory's inode number; and the remaining maps store inode metadata (mtime, mode, and size). Both `InoT` and `U64T` are 64-bit integer types, and `NameT` is a string type.

The `FSSpec` data structure itself places only weak constraints on the logical layout of YminLFS. For example, it does not rule out layouts in which an inode `d` contains an inode `f` according to the `childmap`, but `f` is not contained in `d` according to the `parentmap`. The `FSSpec` specification disallows such invalid layouts with a *well-formedness invariant*:

```
def invariant(self):
 ino, name = InoT(), NameT()
 return ForAll([ino, name], Implies(
  self._childmap[(ino, name)] > 0,
  self._parentmap[self._childmap[(ino, name)]] == ino))
```

The invariant says that the parent and child mappings of valid (positive) inode numbers agree with each other. Both `ForAll` and `Implies` are built-in logical operators.

FILE SYSTEM OPERATIONS. Given our logical layout, we can now specify the desired behavior of file system operations. Read-only operations, such as `lookup` and `stat`, are easy to define:

```
def lookup(self, parent, name):
  ino = self._childmap[(parent, name)]
  return ino if ino > 0 else -errno.ENOENT

def stat(self, ino):
  return Stat(size=self._sizemap[ino],
              mode=self._modemap[ino],
              mtime=self._mtimemap[ino])
```

Operations that modify the file system are more complex, as they involve updating the state of the abstract maps. For example, to add a new file to a given directory, `mknod` needs to update all abstract maps as follows:

```
def mknod(self, parent, name, mtime, mode):
  # Name must not exist in parent.
  if self._childmap[(parent, name)] > 0:
    return -errno.EEXIST

  # The new ino must be valid & not already exist.
  ino = InoT()
  assertion(ino > 0)
  assertion(Not(self._parentmap[ino] > 0))

  with self.transaction():
    # Update the directory structure.
    self._childmap[(parent, name)] = ino
    self._parentmap[ino] = parent
    # Initialize inode metadata.
    self._mtimemap[ino] = mtime
    self._modemap[ino]  = mode
    self._sizemap[ino]  = 0

  return ino
```

The `InoT()` constructor returns an abstract inode number, which is constrained to be valid (i.e., positive) and not present in any directory. The changes to the file system are wrapped in a transaction to ensure that they happen atomically or not at all (if the system crashes).

STATE EQUIVALENCE PREDICATE. The last part of our YminLFS specification defines what it means for a given file system state to be correct:

```
def equivalence(self, impl):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], And(
    self.lookup(ino, name) == impl.lookup(ino, name),
    Implies(self.lookup(ino, name) > 0,
      self.stat(self.lookup(ino, name)) ==
      impl.stat(impl.lookup(ino, name)))))
```

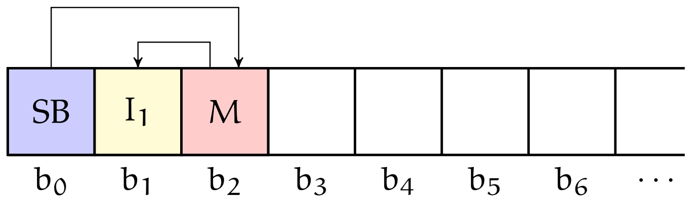
In particular, we require a correct implementation to contain the same files as the abstract data structure, and each file to have the same metadata as its abstract counterpart.

PUTTING IT ALL TOGETHER. With our toy specification completed, we now highlight two key features of the Yggdrasil specification approach. First, Yggdrasil specifications are free of implementation details and are therefore reusable. The `FSSpec` data structure does not mandate any particular on-disk layout, nor does it force the implementation to be, for example, a log-structured file system. In fact, our Yxv6 journaling file system is built on top of an extension of this specification (see Section 3.3).

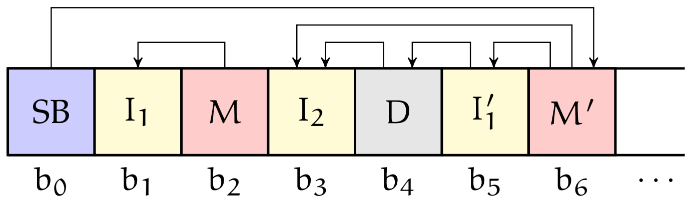
Second, Yggdrasil specifications are both succinct and expressive. For example, the specification of `mknod` provides two deep properties in just a few lines of code: crash-free functional correctness (i.e., a file will be created with the correct metadata if there is no crash); and crash safety (i.e., file creation is all-or-nothing even in the face of crashes).

3.1.2 *Implementation*

To implement a file system in Yggdrasil, the programmer needs to choose a *disk model*, write the code for each specified operation, and write the *consistency invariants* for the on-disk layout. We describe the disk model next, followed by a brief overview of the implementation and consistency invariants for YminLFS. We omit full implementation details (200 lines of Python) for space reasons.



(a) The initial disk state of an empty root directory.



(b) The disk state after adding one file.

Figure 2: YminLFS's on-disk layout. SB is the superblock; I denotes an inode block; M denotes an inode mapping block; D denotes a data block; arrows denote pointers.

DISK MODEL. Yggdrasil provides several disk models: YminLFS (as well as Yxv6) uses the asynchronous disk model; we will use a synchronous one in Section 3.4. The asynchronous disk model specifies a block device that has an unbounded volatile cache and allows arbitrary reordering. Its interface includes the following operations:

- d.`write`(a, v): write a data block v to disk address a;
- d.`read`(a): return a data block at disk address a; and
- d.`flush`$()$: flush the disk cache.

This disk model is trusted to be a correct specification of the underlying physical disk, as we discuss in Section 3.3.2. Unless otherwise specified, we assume 64-bit block addresses and 4 KB blocks. We also assume that a single block read/write is atomic, similar to prior work [23, 122].

A LOG-STRUCTURED FILE SYSTEM. YminLFS is implemented as a log-structured file system that works in a copy-on-write fashion. In particular, it does not overwrite existing blocks (except for the superblock in block zero); it has no garbage collection; and it simply fails when it runs out of blocks, inodes, or directory entries. Its interface provides a `mkfs` operation for initializing the disk, as well as the operations for reading and modifying the file system state that we specified in Section 3.1.1.

The `mkfs` operation initializes the disk as shown in Figure 2a. The effect of the operation is to create a file system with a single empty root directory. This involves writing three blocks: the superblock, an inode I_1 for the root directory, and an inode mapping M that stores the mapping from inode numbers to block numbers. After initialization, M has one entry, $1 \mapsto b_1$, and I_1 points to no data blocks, as the root directory is empty. The superblock points to M, and it stores two additional counters: the next available inode number i (which is initialized to 2 since the root is 1) and the next available block number b (which is initialized to 3).

To add a file to the root directory, `mknod` changes the disk state from Figure 2a to Figure 2b, as follows:

1. add an inode block I_2 for the new file;
2. add a data block D for the root directory, which now has one entry that maps the name of the new file to its inode number 2;
3. add an inode block I'_1 for the updated root directory, which points to its data block D;
4. add an inode mapping block M', which has two entries: $1 \mapsto b_5$ and $2 \mapsto b_3$;
5. finally, update the superblock SB to point to the latest inode mapping M'.

Since the disk can reorder these updates, `mknod` must issue disk flushes to be crash-safe. For example, if there is no flush between the last two writes (steps 4 and 5), the disk can reorder them; if the system crashes in between the reordered writes, the superblock will point to garbage data in b_6, resulting in corrupted YminLFS state. For now, we assume a naïve but correct implementation of `mknod` that inserts five flushes, one after each write. In Section 3.1.4, we will use the Yggdrasil optimizer to remove the first three flushes.

CONSISTENCY INVARIANTS. A consistency invariant for a file system implementation is analogous to the well-formedness invariant for its specification—it is a predicate that determines whether a given disk state corresponds to a valid file-system image. Yggdrasil uses consistency invariants for two purposes: push-button verification and run-time checking in the style of `fsck` [64, 104]. For verification, Yggdrasil checks that the invariant holds for the initial file system state right after `mkfs`; in addition, it assumes the consistency invariant as part of the precondition for each operation, and checks that the invariant holds as part of the postcondition. Once the implementation is verified, Yggdrasil can optionally generate an `fsck`-like checker from these invariants (though the checker cannot repair corrupted file systems). Such a checker is useful even for a bug-free file system, as hardware failures and bugs in other parts of the system can damage the file system [125].

The YminLFS consistency invariant constrains three components of the on-disk layout (Figure 2): the superblock *SB*, the inode mapping block M, and the root directory data block D. The superblock constraint requires the next available inode number i to be greater than 1, the next available block number b to be greater than 2, and the pointer to M to be both positive and smaller than b. The inode mapping constraint ensures that M maps each inode number in range $(0, i)$ to a block number in range $(0, b)$. Finally, the root directory constraint requires D to map file names to inode numbers in range $(0, i)$. These three constraints are all Yggdrasil needs to verify YminLFS (see Section 3.1.3).

3.1.3 *Verification*

To verify that the YminLFS implementation (Section 3.1.2) satisfies the `FSSpec` specification (Section 3.1.1), Yggdrasil uses the Z3 solver [109] to prove a two-part crash refinement theorem (Section 3.2). The first part of the theorem deals with crash-free executions. It requires the implementation and specification to behave alike in the absence of crashes: if both YminLFS and `FSSpec` start in equivalent and consistent states, they end up in equivalent and consistent states. The verifier defines equivalence using the specification's `equivalent` predicate (Section 3.1.1), and consistency using the implementation's consistency invariants (Section 3.1.2).

The second part of the theorem deals with crashing executions. It requires the implementation to exhibit no more crash states (disk states after a crash) than the specification: each possible state of the YminLFS implementation (including states caused by crashes and reordered writes) must be equivalent to *some* crash state of `FSSpec`.

COUNTEREXAMPLES. If there is any bug in the implementation or consistency invariants, the verifier will generate a *counterexample* to help programmers understand the bug. A counterexample consists of a concrete trace of the implementation that violates the crash refinement theorem. As an example, consider the potential missing flush bug described in Section 3.1.2. If we remove the flush between the last two writes in the implementation of `mknod`, Yggdrasil outputs the following counterexample:

```
# Pending writes
lfs.py:167 mknod write(new_imap_blkno, imap)

# Synchronized writes
lfs.py:148 mknod write(new_blkno, new_ino)
lfs.py:154 mknod write(new_parentdata, parentdata)
lfs.py:160 mknod write(new_parentblkno, parentinode)
lfs.py:170 mknod write(SUPERBLOCK, sb)

# Crash point
[..]
lfs.py:171 mknod flush()
```

The output describes the bug by showing the point at which the system crashes and the list of writes pending in the cache (along with their source code locations). In this example, the write of the new inode mapping block (step 4 above) is still pending, but the write to update the superblock to point to that block (step 5) has reached the disk, corrupting YminLFS's state.

The visualization of "pending" and "synchronized" writes in the counterexample is specific to the asynchronous disk model; one can extend Yggdrasil with new disk models and customized visualizations.

Our initial YminLFS implementation contained two other bugs: one in the `lookup` logic and one in the data layout. Neither of the bugs appeared during testing runs. Both bugs were found by the verifier in a matter of seconds, and we quickly localized and fixed them by examining the resulting counterexamples.

PROOFS. If the Yggdrasil verifier finds no counterexamples to the crash refinement theorem, then none exist, and we have obtained a *proof* of correctness. In particular, the crash refinement theorem holds for all disks with up to 2^{64} blocks, and for every trace of file system operations, regardless of its length. After we fixed the bugs in our initial YminLFS implementation, the verifier proved its correctness in under 30 seconds.

It is worth noting that the theorem holds if the file system is the only user of the disk. For instance, it does *not* hold if an adversary corrupted the file system image by directly modifying the disk. To address this issue, one can run `fsck` generated by Yggdrasil, which guarantees to detect any such inconsistencies.

3.1.4 *Optimizations and compilation*

As described in Section 3.1.2, YminLFS's `mknod` implementation uses five disk flushes. Yggdrasil provides a greedy optimizer that tries to remove every disk flush and re-verify the code. Running the optimizer on the `mknod` code removes three out of the five flushes within three minutes, while still guaranteeing correctness.

The optimized and verified YminLFS implementation, which is in Python, is executable but slow. Yggdrasil invokes the Cython compiler [10] to generate C code from Python for better performance. It also provides a small bridge to connect the generated C code to FUSE [43]. The result is a single-threaded user-space file system.

3.1.5 *Summary*

We have demonstrated how to specify, implement, debug, verify, optimize, and execute the YminLFS file system using Yggdrasil. Compared to previous file system verification work, push-button verification eases the proof burden and enables automated features such as visualizing bugs and optimizing code.

Since there is no need to manually prove or annotate implementation code when using Yggdrasil, the verification effort is spent mainly on writing the specification and coming up with consistency invariants about the on-disk data format. We find the counterexample visualizer useful for finding bugs in these two parts.

The trusted computing base (TCB) includes the file system specification, Yggdrasil's verifier, visualizer, and compiler (but not the optimizer), their dependencies (i.e., the Z3 solver, Python, and gcc), as well as FUSE and the Linux kernel. See Section 3.5 for discussion on limitations.

3.2 THE YGGDRASIL ARCHITECTURE

In Yggdrasil, the core notion of correctness is crash refinement. This section gives a formal definition of crash refinement, and describes how Yggdrasil's components use this definition to support verification, counterexample visualization, and optimization.

3.2.1 *Reasoning about systems with crashes*

In Yggdrasil, programmers write both specifications and implementations (referred to as "systems" in this section) as state machines: each system comprises a state and a set of operations that transition the state. A transition can occur only if the system is in a consistent state, as determined by its consistency invariant $\mathcal{I}$. This invariant is a predicate over the system's state, indicating whether it is consistent or corrupted; see Section 3.1.2 for an example.

Consider a specification F_0 and an implementation F_1. Our goal is to show that F_1 is correct with respect to F_0. Since both systems are state machines, a strawman definition of correctness is that they transition in lock step (i.e., bisimulation): starting from equivalent consistent states, if the same operation is invoked on both systems, they will transition to equivalent consistent states (where equivalence between states is defined by a system-specific predicate). However, this bisimulation-based definition is too strong for systems that interact with external storage, as it does not account for non-determinism from disk reorderings, crashes, or recovery.

To address this shortcoming, we introduce *crash refinement* as a new definition of correctness. At a high level, crash refinement says that F_1 is correct with respect to F_0 if, starting from equivalent consistent states and invoking the same operation on both systems, *any* state produced by F_1 is equivalent to *some* state produced by F_0. To formalize this intuition, we define the behavior of a system in the presence of crashes, formalize crash refinement for individual operations, and extend the resulting definition to entire systems.

SYSTEM OPERATIONS. We model the behavior of a system operation with a function f that takes three inputs:

- its current state s;
- an external input $\mathbf{x}$, such as data to write; and
- a crash schedule $\mathbf{b}$, which is a set of boolean values denoting the occurrence of crash events.

Applying f to these inputs, written as $f(s, \mathbf{x}, \mathbf{b})$, produces the next state of the system.

As a concrete example, consider a single disk write operation that writes value v to disk address a. The external input to the write operation's function f_w is the pair (a, v). The state s is the disk content before the write; $s(a)$ gives the old value at the address a. The asynchronous disk model in Yggdrasil generates a pair of boolean values $(\mathit{on}, \mathit{sync})$ as the crash schedule. The *on* value indicates whether the write operation completed successfully by storing its data into the volatile cache. The *sync* value indicates whether the write's effect has been synchronized from the volatile cache to stable storage. After executing the write operation, the disk is updated to contain v at the address a only if both *on* and *sync* are true, and left unchanged otherwise (e.g., the system crashed before completing the write, or before synchronizing it to stable storage):

$$f_w(s, \mathbf{x}, \mathbf{b}) = s[a \mapsto \texttt{if } \mathit{on} \wedge \mathit{sync} \texttt{ then } v \texttt{ else } s(a)],$$
$$\text{where } \mathbf{x} = (a, v) \text{ and } \mathbf{b} = (\mathit{on}, \mathit{sync}).$$

CRASH REFINEMENT. To define crash refinement for a given schedule, we start from a special case where write operations always complete and their effects are synchronized to disk. That is, the crash schedule is the constant vector ***true***. Let $s_0 \sim s_1$ denote that s_0 and s_1 are equivalent states according to a user-defined equivalence relation (as in Section 3.1.1). We write $s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1$ to say that s_0 and s_1 are equivalent and consistent according to their respective system invariants $\mathcal{I}_0$ and $\mathcal{I}_1$:

$$s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1 \triangleq \mathcal{I}_0(s_0) \wedge \mathcal{I}_1(s_1) \wedge s_0 \sim s_1.$$

With a crash-free schedule ***true***, two functions f_0 and f_1 are equivalent if they produce equivalent and consistent output states when given the same external input $\mathbf{x}$, as well as equivalent and consistent starting states:

Definition 1 (Crash-free equivalence)**.** Given two functions f_0 and f_1 with their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, we say f_0 and f_1 are *crash-free equivalent* if the following holds:

$$\forall s_0, s_1, x.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, x, \textbf{\textit{true}}) \text{ and } s_1' = f_1(s_1, x, \textbf{\textit{true}}).$$

Next, we allow for the possibility of crashes. We say that f_1 is correct with respect to f_0 if, for any crash schedule, the state produced by f_1 with that schedule is equivalent to a state produced by f_0 with *some* schedule:

Definition 2 (Crash refinement without recovery)**.** Function f_1 is a *crash refinement (without recovery)* of f_0 if (1) f_0 and f_1 are crash-free equivalent and (2) the following holds:

$$\forall s_0, s_1, x, \mathbf{b}_1.\ \exists \mathbf{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, x, \mathbf{b}_0) \text{ and } s_1' = f_1(s_1, x, \mathbf{b}_1).$$

Finally, we consider the possibility that the system may run a *recovery function* upon reboot. A recovery function r is a system operation (as defined above) that takes no external input (as it is executed when the system starts). It should also be *idempotent*: even if the system crashes during recovery and re-runs the recovery function many times, the resulting state should be the same once the recovery is complete.

Definition 3 (Recovery idempotence)**.** A recovery function r is idempotent if the following holds:

$$\forall s, \mathbf{b}.\ r(s, \textbf{\textit{true}}) = r(r(s, \mathbf{b}), \textbf{\textit{true}}).$$

Note that this definition accounts for multiple crash-reboot cycles during recovery, by repeated application of the idempotence definition on each intermediate crash state $r(s, \mathbf{b})$, $r(r(s, \mathbf{b}), \mathbf{b'})$, ..., where $\mathbf{b}$, $\mathbf{b'}$, ... are the schedules for each crash during recovery.

Definition 4 (Crash refinement with recovery)**.** Given two functions f_0 and f_1, their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, and a recovery function r, f_1 with r is a *crash refinement* of f_0 if (1) f_0 and f_1 are crash-free equivalent; (2) r is idempotent; and (3) the following holds:

$$\forall s_0, s_1, x, \mathbf{b}_1.\ \exists \mathbf{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, x, \mathbf{b}_0) \text{ and } s_1' = r(f_1(s_1, x, \mathbf{b}_1), \textbf{\textit{true}}).$$

Furthermore, systems may run background operations that do not change the externally visible state of a system (i.e., no-ops), such as garbage collection.

Definition 5 (No-op)**.** Function f with a recovery function r is a *no-op* if (1) r is idempotent, and (2) the following holds:

$$\forall s_0, s_1, x, \mathbf{b}_1.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_1' = r(f(s_1, x, \mathbf{b}_1), \textbf{\textit{true}}).$$

With per-function crash refinement and no-ops, we can now define crash refinement for entire systems.

Definition 6 (System crash refinement)**.** Given two systems F_0 and F_1, and a recovery function r, F_1 is a *crash refinement* of F_0 if every function in F_1 with r is either a crash refinement of the corresponding function in F_0 or a no-op.

The rest of this section will describe Yggdrasil's components based on the definition of crash refinement.

3.2.2 *The verifier*

Given two file systems, F_0 and F_1, Yggdrasil's verifier checks that F_1 is a crash refinement of F_0 according to Definition 6. To do so, the verifier performs symbolic execution [21, 76] for each operation $f_i \in F_i$ to obtain an SMT encoding of the operation's output, $f_i(s_i, x, \mathbf{b}_i)$, when applied to a symbolic input x (represented as a bitvector), symbolic disk state s_i (represented as an uninterpreted function over bitvectors), and symbolic crash schedule $\mathbf{b}_i$ (represented as booleans). It then invokes the Z3 solver to check the validity of either the no-op identity (Definition 5) if f_1 is a no-op, or else the per-function crash refinement formula (Definition 4) for the corresponding functions $f_0 \in F_0$ and $f_1 \in F_1$.

To capture *all* execution paths in the SMT encoding of $f_i(s_i, x, \mathbf{b}_i)$, the verifier adopts a "self-finitizing" symbolic execution scheme [151], which simply unrolls loops and recursion *without* bounding the depth. Since this scheme will fail to terminate on non-finite code, the verifier requires file systems to be implemented in a finite way: for instance, loops must be bounded [152]. In our experience (further discussed in Section 3.3), the finiteness requirement does not add much programming burden.

To prove the validity of the per-function crash refinement formula, the verifier uses Z3 to check if the formula's negation is unsatisfiable. If so, the result is a proof that f_1 is a crash refinement of f_0. Otherwise, Z3 produces a *model* of the formula's negation, which represents a concrete counterexample to crash refinement: disk states s_0 and s_1, an input x, and a crash schedule $\mathbf{b}_1$, such that $s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1$ but there is no crash schedule $\mathbf{b}_0$ that satisfies $f_0(s_0, x, \mathbf{b}_0) \sim_{\mathcal{I}_0,\mathcal{I}_1} f_1(s_1, x, \mathbf{b}_1)$.

Checking the satisfiability of the negated crash refinement formula in Definition 4 requires reasoning about quantifiers. In general, such queries are undecidable. In our case, the problem is decidable because the quantifiers range over finite domains, and the formula is expressed in a decidable combination of decidable theories (i.e., equality with uninterpreted functions and fixed-width bitvectors) [157]. Moreover, Z3 can solve this problem in practice because the crash schedule $\mathfrak{b}_0$, which is a set of boolean variables, is the only universally quantified variable in the negated formula. As many file system specifications have simple semantics, the crash schedule $\mathfrak{b}_0$ has few boolean variables—often only one (e.g., the transaction in Section 3.1.1)—which makes the reasoning efficient.

The verifier's symbolic execution engine supports all regular Python code with concrete (i.e., non-symbolic) values. For symbolic values, it supports booleans, fixed-width integers, maps, and lists of concrete length, as well as regular control flow including conditionals and loops, but no exceptions or coroutines. It does not support symbolic execution into C library code.

3.2.3 *The counterexample visualizer*

To make counterexamples to validity easier to understand, Yggdrasil provides a visualizer for the asynchronous disk model. Given a counterexample model of the formula in Definition 4, the visualizer produces concrete disk event traces (e.g., see Section 3.1.3) as follows. First, it uses the crash schedule $\mathfrak{b}_1$ to identify the boolean variable *on* that indicates where the system crashed, and relates that location to the implementation source code with a stack trace. Second, it evaluates the boolean *sync* variables that indicate whether a write is synchronized to disk, and prints out the pending writes with their corresponding source locations to help identify unintended reorderings. Yggdrasil also allows programmers to supply their own plugin visualizer for data structures specific to their file system images. We found this facility useful when developing YminLFS and Yxv6.

3.2.4 *The optimizer*

The Yggdrasil optimizer improves the run-time performance of implementation code. Yggdrasil treats the optimizer as untrusted and re-verifies the optimized code it generates. This simple design, made possible by push-button verification, allows programmers to plug in custom optimizations without the burden of supplying a correctness proof. We provide one built-in optimization that greedily removes disk flush operations (see Section 3.1.4), implemented by rewriting the Python abstract syntax tree.

3.3 THE YXV6 FILE SYSTEM

The section describes the design, implementation, and verification of the Yxv6 journaling file system. At a high level, verifying the correctness of Yxv6 requires Yggdrasil to obtain an SMT encoding of both the specification and implementation through symbolic execution, and to invoke an SMT solver to prove the crash refinement theorem. A simple approach, used by YminLFS in Section 3.1, is to directly prove crash refinement between the entire file system specification and implementation. However, the complexity of Yxv6 makes such a proof intractable for state-of-the-art SMT solvers. To address this issue, Yxv6 employs a modular design enabled by crash refinement to scale up SMT reasoning.

3.3.1 *Design overview*

Yxv6 uses crash refinement to achieve scalable SMT reasoning in three steps. First, to reduce the size of SMT encodings, Yxv6 stacks five layers of abstraction, each consisting of a specification and implementation, starting with an asynchronous disk specification (Section 3.3.2). We use Yggdrasil to prove crash refinement theorems for each layer, showing that each correctly implements its specification. Upper layers then use the specifications of lower layers, rather than their implementations, in order to accelerate verification. This layered approach effectively bounds the reasoning to a single layer at a time.

Second, many file system operations touch only a small part of the disk. To allow the SMT solver to exploit this locality, Yxv6 explicitly uses multiple separate disks rather than one. For example, by storing the free bitmap on a separate disk, the SMT solver can easily infer that updating it does not affect the rest of the file system. We then prove crash refinement from this multi-disk system to a more space-efficient file system that uses only a single disk (Section 3.3.3). The result of these first two steps is Yxv6+sync, a *synchronous* file system that commits a transaction for each system call (by forcing the log to disk), similar to xv6 [32] and FSCQ [23].

Finally, for better run-time performance, we implement an optimized variant of Yxv6+sync that groups multiple system calls into one transaction [57] and commits only when the log is full or upon `fsync`. We prove the resulting file system, called Yxv6+group_commit, is a crash refinement of Yxv6+sync with a more relaxed crash consistency model (Section 3.3.4).

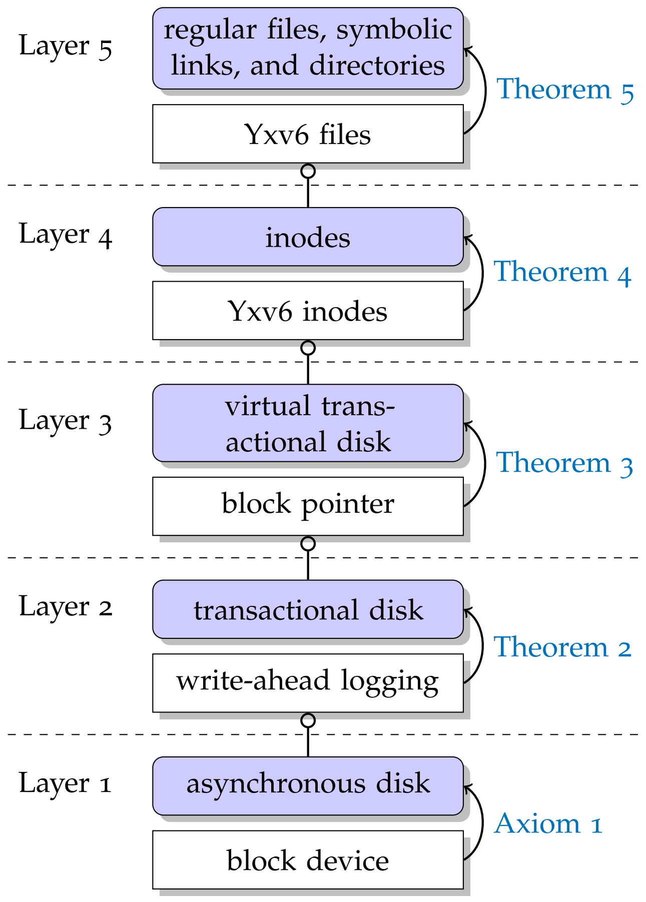


Figure 3: The stack of layers of Yxv6. Within each layer, a shaded box represents the specification; a (white) box represents the implementation; and the implementation is a crash refinement of its specification, denoted using an arrow. Each implementation (except for the lowest layer) builds on top of a specification from the layer below, denoted using a circle.

3.3.2 *Stacking layers of abstraction*

Figure 3 shows the five abstraction layers of Yxv6. Each layer consists of a specification and an implementation that is written using a lower-level specification. We describe each of these layers in turn.

LAYER 1: ASYNCHRONOUS DISK. The lowest layer of the stack is a specification of an asynchronous disk. This specification comprises the asynchronous disk model we used in Section 3.1.2 to implement YminLFS. Since the implementation of a physical block device is opaque, we assume the specification correctly models the block device (i.e., the specification is more conservative and allows more behavior than real hardware), as follows:

Axiom 1. A block device is a crash refinement of the asynchronous disk specification.

LAYER 2: TRANSACTIONAL DISK. The next layer introduces the abstraction of a transactional disk, which manages multiple separate data disks, and offers the following operations:

- `d.begin_tx()` starts a transaction;
- `d.commit_tx()` commits a transaction;
- `d.write_tx`(j, a, v) adds to the current transaction a write of value v to address a on disk j; and
- `d.read`(j, a) returns the value at address a on disk j.

The specification says that operations executed within the same transaction are atomic (i.e., all-or-nothing) and sequential (i.e., transactions cannot be reordered).

The implementation uses the standard write-ahead logging technique [57, 106]. It uses one asynchronous disk (from layer 1) for the log, and a set of asynchronous disks for data. Using a single transactional disk to manage multiple data disks allows higher layers to separate writes within a transaction (e.g., updates to data and inode blocks will not interfere), which helps scale SMT reasoning; Section 3.3.3 refines the multiple disks to one.

The implementation is parameterized by the transaction size limit k (i.e., the maximum number of writes in one transaction). The log disk uses a fixed number of blocks, determined by k, as a header to store log entry addresses, and the remaining blocks to store log entry data. The first entry in the first header block is a counter of log entries; the consistency invariant for the transactional disk layer says that this counter is always zero after recovery. The Yxv6+sync file system sets $k = 10$, while Yxv6+group_commit sets $k = 511$. For each of these settings, we prove the following theorem:

Theorem 2. The write-ahead logging implementation is a crash refinement of the transactional disk specification.

LAYER 3: VIRTUAL TRANSACTIONAL DISK. The specification of the virtual transactional disk is similar to that of the transactional disk, but instead uses 64-bit *virtual disk addresses* [73]. Each virtual address can be mapped to a physical disk address or unmapped later; reads and writes are valid for mapped addresses only. We will use this abstraction to implement inodes in the upper layer.

The virtual transactional disk implementation uses the standard block pointers approach. It uses one transactional disk managing at least three data disks: one to store the free block bitmap, another to store direct block pointers, and the third to store both data and singly indirect block pointers (higher layers will add additional disks). The free block bitmap disk stores only one bit in each of its blocks, which simplifies SMT reasoning but wastes disk space; Section 3.3.3 will refine it to a more space-efficient version.

The implementation relies on two consistency invariants: (1) the mapping from virtual disk addresses to physical disk addresses is injective (i.e., each physical address is mapped at most once), and (2) if a virtual disk address is mapped to physical address a, the a^{th} bit in the block bitmap must be marked as used. We use these invariants to prove the following theorem:

Theorem 3. The block pointer implementation is a crash refinement of the virtual transactional disk specification.

LAYER 4: INODES. The fourth layer introduces the abstraction of inodes. Each inode is uniquely identified using a 32-bit inode number. The specification maps an inode number to 2^{32} blocks, and to a set of metadata such as `size`, `mtime`, and `mode`.

The implementation is straightforward thanks to the virtual transactional disk specification. It simply splits the 64-bit virtual disk address space into 2^{32} ranges, and each inode takes one range, which has 2^{32} "virtual" blocks, similar to NVMFS/DFS [73]. Inode metadata resides on a separate disk managed by the virtual transactional disk (which now has four data disks). There are no consistency invariants in this layer. We prove the following theorem:

Theorem 4. The Yxv6 inode implementation is a crash refinement of the inode specification.

LAYER 5: FILE SYSTEM. The top layer of the file system is an extended version of `FSSpec` given in Section 3.1, with regular files, directories, and symbolic links.

The implementation builds on top of the inode specification, using a separate inode bitmap disk and another for orphan inodes. Both are managed by the virtual transactional disk (which now has six data disks plus the log disk, giving a total of seven disks). There are two consistency invariants: (1) if an inode is not marked as used in the inode bitmap disk, its size must be zero in the metadata; and (2) if an inode has n blocks, no "virtual" block larger than n is mapped. Using these invariants, we prove the final crash refinement theorem:

Theorem 5. The Yxv6 implementation of files is a crash refinement of the specification of regular files, symbolic links, and directories.

FINITIZATION. The Yggdrasil verifier requires Yxv6 operations to be finite, as mentioned in Section 3.2.2. Most file system operations satisfy this requirement, as they use only a small number of disk reads and writes. For example, moving a file involves updating only the source and destination directories. However, there are two exceptions.

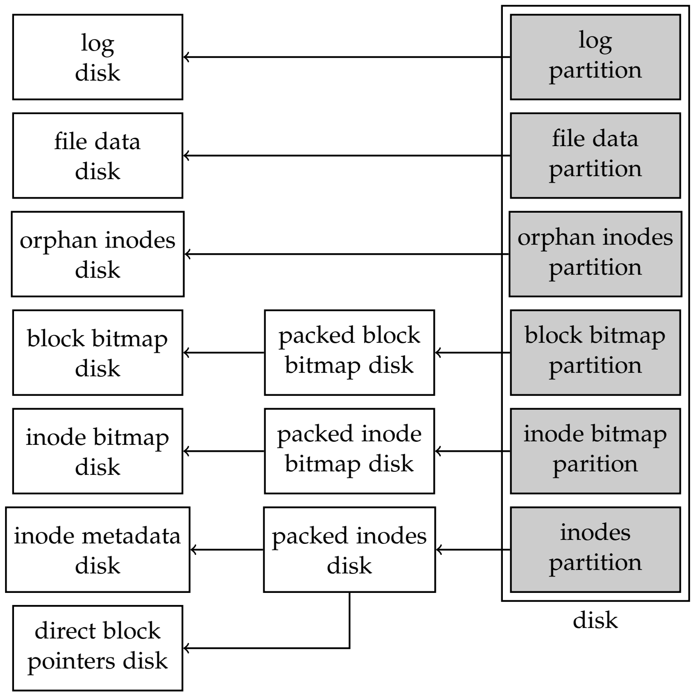


Figure 4: The refinement of disk layout of the Yxv6 file system, from multiple disks to a single disk. The arrows $A \leftarrow B$ denote that B is a crash refinement of A.

First, search-related procedures, such as finding a free bit in a bitmap, may need to read many blocks. We choose *not* to verify the bit-finding algorithm, but instead adopt the idea of validation [123, 139, 144] to implement such search algorithms. The validator, which we do verify, simply checks that an index returned by the search is indeed marked free in the bitmap and if not, fails the operation with an error code. We use similar strategies for directory entry lookup. This approach allows us to treat search procedures as a black box, absolving the SMT solver from the need to reason about the many paths through the algorithm.

The second case is unlinking a file, as freeing all its data blocks needs to write potentially many blocks. To finitize this operation, our implementation simply moves the inode of the file into a special orphan inodes disk, which is a finite operation, and relies on a separate garbage collector to reclaim the data blocks at a later time. We further prove that reclamation is a no-op (as per the definition in Section 3.2.1), as freeing a block referenced by the orphan inodes disk does not affect the externally visible state of the file system. We will summarize the trade-offs of validation in Section 3.3.5.

3.3.3 *Refining disk layouts*

Theorem 5 gives a file system that runs on seven disks: the write-ahead log, the file data, the block and inode bitmaps for managing free space, the inode metadata, the direct block pointers, and the orphan inodes. Using separate disks scales SMT reasoning, but it has two downsides. First, the two bitmaps use only one bit per block and the inode metadata disk stores one inode per block, wasting space. Second, requiring seven disks makes the file system difficult to use. We now prove with crash refinement that it is correct to pack these disks into one disk (Figure 4) similar to the xv6 file system [32].

Intuitively, it is correct to pack multiple blocks that store data sparsely into one with a dense representation, because the packed disk has the same or fewer possible disk states. For instance, bitmap disks used in Section 3.3.2 store one bit per block; the n-th bit of the bitmap is stored in the lowest bit of block n. On the other hand, a *packed* bitmap disk stores $4\,\text{KB} \times 8 = 2^{15}$ bits per block, and the n-th bit is stored in bit $n \bmod 2^{15}$ of block $n/2^{15}$. Clearly, using the packed bitmap is a crash refinement of the sparse one. The same holds for using packed inodes. Similarly, a single disk with multiple non-overlapping partitions exhibits fewer states than multiple disks; for example, a flush on a single disk will flush all the partitions, but not for multiple disks. Combining these packing steps, we prove the following theorem:

Theorem 6. The Yxv6 implementation using seven non-overlapping partitions of one asynchronous disk, with packed bitmaps and inodes, is a crash refinement of that using seven asynchronous disks.

3.3.4 *Refining crash consistency models*

Theorem 6 gives a *synchronous* file system that commits a transaction for each system call. This file system, which we call Yxv6+sync, incurs a slowdown as it flushes the disk frequently (see Section 3.7 for performance evaluation). The Yxv6+group_commit file system implements a more relaxed crash consistency model [18, 122]. Unlike Yxv6+sync, its write-ahead logging implementation groups multiple transactions together [57].

Intuitively, doing a single combined transaction produces fewer possible disk states compared to two separate transactions, as in the latter scheme the system can crash in between the two and expose the intermediate state. We prove the following theorem:

Theorem 7. Yxv6+group_commit is a crash refinement of Yxv6+sync.

3.3.5 *Summary of design trade-offs*

Unlike conventional journaling file systems, the first Yxv6 design in Section 3.3.2 uses multiple disks. To decide the number of disks, we adopt a simple guideline: whenever a part of the disk is *logically* separate from the rest of the file system, such as the log or the free bitmap, we assign a separate disk for that part. In our experience, this is effective in scaling up SMT reasoning.

Yxv6's final on-disk layout closely resembles that of the xv6 and FSCQ file systems. One notable difference is that Yxv6 uses an orphan inodes partition to manage files that are still open but have been unlinked, similarly to the orphan inode list [63] in ext3 and ext4. This design ensures correct atomicity behavior of `unlink` and `rename`, especially when running with FUSE, which xv6 and FSCQ do not guarantee.

Another difference to FSCQ is that Yxv6 uses validation instead of verification in managing free blocks and inodes. Although the resulting allocator is safe, it does not guarantee that block or inode allocation will succeed when there is enough space, treating such failures as a quality-of-service issue.

3.4 BEYOND FILE SYSTEMS

Although we designed Yggdrasil for writing verified file systems, the idea of crash refinement generalizes to applications that use disks in other ways. This section describes two examples: Ycp, a file copy utility; and Ylog, a persistent log data structure.

THE YCP FILE COPY UTILITY. Like the Unix `cp` utility, Ycp copies the contents of one file to another. Unlike `cp`, it has a formal specification: if the copy operation succeeds, the file system is updated so that the target file contains the same data as the source file; if it fails due to a system crash or an invalid target (e.g., a directory or a symbolic link), the file system is unchanged.

The implementation of Ycp uses the Yxv6 file system specification (Figure 3). It follows a common atomicity pattern: (1) create a temporary file, (2) write the source data to it, and (3) rename it to atomically create the target file. There is no consistency invariant as Ycp uses file system operations and is independent of disk layout.

We verify that the implementation of Ycp is a crash refinement of its specification using Yggdrasil. This shows that Yggdrasil and Yxv6's specification are useful for reasoning about application-level correctness.

THE YLOG PERSISTENT LOG. Ylog is a verified implementation of the persistent log from the Arrakis operating system [121]. The

component	specification	implementation	consistency inv
Yxv6	250	1,500	5
YminLFS	25	150	5
Ycp	15	45	0
Ylog	35	60	0
infrastructure	–	1,500	–
FUSE stub	–	250	–

Figure 5: Lines of code for the Yggdrasil toolkit and storage systems built using it, excluding blank lines and comments.

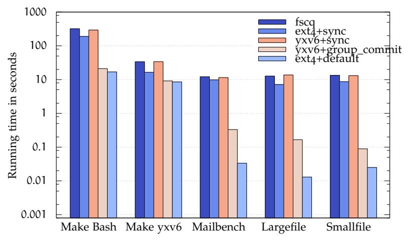


Figure 6: Performance of file systems on an SSD, in seconds (log scale; lower is better).

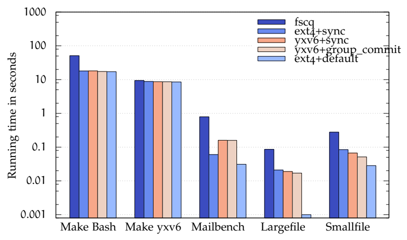


Figure 7: Performance of file systems on a RAM disk, in seconds (log scale; lower is better).

Arrakis log is designed to provide an efficient storage API with strong atomicity and persistence guarantees. The core logging operation is a multi-block append, which extends an on-disk log with entries that can span multiple blocks. This append operation must appear to be both atomic and immediately persistent, even in the presence of crashes.

The Arrakis persistent log was originally designed to run on top of an LSI Logic MegaRAID SAS-3 3108 RAID controller with a battery-backed cache. We therefore chose to implement Ylog on top of a *synchronous* disk model, which does not reorder writes and matches the behavior of the RAID controller. Ylog uses the same on-disk layout as Arrakis: the first block (i.e., superblock) contains metadata, such as the number of entries and a pointer to the end of the log, followed by blocks that contain the data of each entry.

When comparing Ylog's implementation with that of Arrakis, we discovered two bugs in the Arrakis persistent log: its crash recovery logic was *not* idempotent, and the log could end up with garbage data if the system crashed again during recovery. The bugs were reported to and confirmed by the Arrakis developers.

3.5 DISCUSSION

This section discusses the limitations of Yggdrasil, as well as our experience using and designing the toolkit.

LIMITATIONS. Yggdrasil reasons about single-threaded code, so file systems written using Yggdrasil do not support concurrency. Cython [10], Yggdrasil's Python-to-C compiler, is unverified, although we have not yet encountered any bugs in the development.

Yggdrasil relies on SMT solvers for automated reasoning, and is limited to first-order logic. It is less expressive than interactive theorem provers such as Coq or Isabelle, although our experience shows that it is sufficient for writing and verifying file systems like Yxv6 based on crash refinement.

Since the Z3 solver is at the core of Yggdrasil, its correctness is critical. To understand this risk, we ran the Yxv6 verification using every buildable snapshot of the Z3 Git repository over the past three years, a total of 1,417 versions. We also used two other SMT solvers, Boolector [117] and MathSAT 5 [27], for cross-checking. We did not observe any inconsistent results.

The Yxv6 file system lacks several modern file system features, such as extents and delayed allocation in ext4. Compared to hand-written file system checkers, its `fsck` tool is generated by Yggdrasil and guaranteed to detect any violations of consistency invariants, but it cannot repair corrupted file systems.

LESSONS LEARNED. Bitvector operations and reasoning about non-determinism (e.g., crashes) are common in file system implementations. These characteristics motivated us to formulate file system verification as an SMT problem, exploiting the fully automated decision procedures for the theories of bitvectors and uninterpreted functions. In addition, using SMT enables Yggdrasil to produce and visualize counterexamples; we find this ability useful for tracking subtle file system bugs during development, especially corner cases such as overflows and missing flushes [100].

In earlier development of Yggdrasil, we struggled to find a disk representation for scalable SMT reasoning. We explored several approaches, such as a lazy list of symbolic blocks (e.g., EXE [161]) and the theory of arrays, all resulting in a verification bottleneck.

Yggdrasil represents a disk using uninterpreted functions that map a block address and an in-block offset to a 64-bit integer. This two-level map helped to scale up verification. Mapping to 64-bit integers also allowed Yggdrasil to generate efficient C code. The idea of separating logical and physical data representations using crash refinement further reduced the verification time by orders of magnitude. As we will show in Section 3.7, verifying Yxv6+sync's theorems took less than a minute, thanks to Z3's efficient decision procedures, whereas Coq took 11 hours to check the proofs of FSCQ [23] (which has similar features to Yxv6+sync).

Crash refinement requires programmers to design a system as a state machine and implement each operation in a finite way. File systems fit well into this paradigm. We have used crash refinement in several contexts: to stack layers of abstraction, to pack multiple blocks or disks, and to relax crash consistency models. Crash refinement does not require advanced knowledge of program logics (e.g., separation logic [130] in FSCQ), and is amenable to automated SMT reasoning.

BUGS DISCOVERED To our knowledge there have been no bugs discovered in the Yxv6 file system since we released the source in Febuary of 2017. Mohan et al. reported applying their bug finding tool on Yxv6 [107], but did not report finding any bugs. Although this is only anecdotal, we find it encouraging evidence in support of the thesis that it is possible to build correct and reliable software using push-button verification.

3.6 IMPLEMENTATION

Figure 5 lists the code size of the file systems and other storage applications built using Yggdrasil, the common infrastructure code, and the FUSE boilerplate. In total, they consist of about 4,000 lines of Python code.

3.7 EVALUATION

This section uses Yxv6 as a representative example to evaluate file systems built using Yggdrasil. We aim to answer the following questions:

- Does Yxv6 provide end-to-end correctness?
- What is the run-time performance?
- What is the verification performance?

Unless otherwise noted, all experiments were conducted on a 4.0 GHz quad-core Intel i7-4790K CPU running Linux 4.4.0.

CORRECTNESS. We tested the correctness of Yxv6 as follows. First, we ran it on existing benchmarks. Both Yxv6+sync and Yxv6+group_commit passed the `fsstress` tests from the Linux Test Project [85]; they also passed the SibylFS POSIX conformance tests [132], except for incomplete features such as hard links or extended attributes. Second, we used Yxv6 to self-host Yggdrasil's development and our experience is that it is reliable for daily use. Third, we applied the disk block enumerator from the Ferrite toolkit [18] (similar to the Block Order Breaker [122]) to cross-check that the file system state was consistent after a crash and recovery.

To test the correctness of Yxv6's `fsck`, we manually corrupted file system images by overwriting them with random bytes; Yxv6's `fsck` was able to detect corruption in all these cases.

RUN-TIME PERFORMANCE. To understand the run-time performance of Yxv6, we ran a set of five benchmarks similar to those used in FSCQ [23]: compiling the source code of `bash` and Yxv6, running a mail server from the sv6 operating system [28], and the LFS benchmark [134].

We compare the two Yxv6 variants against the verified file system FSCQ and the ext4 file system in two configurations: its default configuration (i.e., `data=ordered`), and with `data=journal+sync` options, which together are similar to Yxv6+sync. Although Yxv6's implementation is closest to xv6, we excluded xv6's performance numbers as it crashed frequently on three benchmarks and did not pass the `fsstress` tests.

Figure 6 shows the on-disk performance with all the file systems running on a Samsung 850 PRO SSD. The y-axis shows total running time in seconds (log scale). We see that Yxv6+sync performs similarly to FSCQ and to ext4's slower configuration. Yxv6+group_commit, which groups several operations into a single transaction, outperforms those file systems by 3–150$\times$ and is on average within 10$\times$ of ext4's default configuration.

To understand the CPU overhead, we repeated the experiments using a RAM disk, as shown in Figure 7. The two variants of Yxv6 have similar performance numbers. They both outperform FSCQ, and are close in performance to ext4 (except for the `largefile` benchmark). We believe the reason is that Yxv6 benefits from Yggdrasil's Python-to-C compiler, while FSCQ's performance is affected by its use of Haskell code extracted from Coq.

VERIFICATION PERFORMANCE. As we mentioned in Section 3.5, the total verification time for Yxv6+sync is under a minute on a single core. It achieved this verification performance due to Z3's efficient SMT solving and the use of crash refinement in the file system.

Verifying Yxv6+group_commit took a longer time, because it is parameterized to use larger transactions (see Section 3.3.2). It finished within 1.6 hours using 24 cores (Intel Xeon 2.2 GHz), approximately 36 hours on a single core.

3.8 RELATED WORK

VERIFIED FILE SYSTEM IMPLEMENTATIONS. Developers looking to build and verify file systems have primarily turned to interactive theorem provers such as Coq [149] and Isabelle [118]. Our approach is most similar to FSCQ [23], a verified crash-safe file system developed in Coq. Their proof shows that after reboot, FSCQ's recovery routines will correctly recover the file system state without data loss. These theorems are stated in *crash Hoare logic*, which extends Hoare logic with support for crash conditions and recovery procedures. Our approach also bears similarities to Flashix [42, 141], another verified crash-safe file system. The Flashix proof consists of several refinements from the POSIX specification layer down to an implementation which can be extracted to Scala. These refinements are proved in the KIV interactive theorem prover in terms of abstract state machines.

Compared to these examples, Yggdrasil's push-button verification substantially lowers the proof burden. Yggdrasil can verify the Yxv6 implementation given only the specifications and five consistency invariants. This ease of verification, together with richer debugging support, also helped us implement several optimizations in Yxv6 that make its performance 3–150× faster than FSCQ and within 10× of ext4.

COGENT [5] takes a different approach to building verified file systems, defining a new restricted language together with a certified compiler to C code. The COGENT language rules out several common sources of errors, such as memory safety and memory leaks, reducing the verification proof burden. We believe Yggdrasil and COGENT to be complimentary: on one hand, COGENT provides certified extraction to C code which could replace Yggdrasil's unverified extraction from Python; on the other hand, Yggdrasil's crash refinement strategy could help COGENT to produce more automated proofs.

FILE SYSTEM SPECIFICATIONS AND CRASH CONSISTENCY. Several projects have developed formal specifications of file systems. SibylFS [132] is an effort to formalize POSIX interfaces and test implementation conformance. But because POSIX file system interfaces underspecify allowed crash behavior, so does the SibylFS formalization. Commuter [28] formalizes the commutativity of POSIX interface calls to study scalability, but as with SibylFS, the formalization does not consider crashes.

Modern file systems adopt various crash recovery strategies, including write-ahead logging (or journaling) [57, 106], log-structured file systems [134], copy-on-write (or shadowing) [17, 133], and soft updates [49, 103]. This diversity complicates reasoning about application-level crash safety. Pillai et al. [122] and Zheng et al. [174] surveyed the crash safety of real-world applications, finding many crash-safety bugs despite extensive engineering effort to tolerate and recover from crashes. Bornholt et al. [18] formalized the crash guarantees of modern file systems as *crash-consistency models*, to help application writers provide crash safety. A formally verified file system can provide these models as an artifact of the verification process. Yggdrasil's crash refinement strategy helps to abstract low-level implementation details out of these application-facing models.

BUG-FINDING TOOLS. Rather than building a new verified file system, several existing projects focus on finding bugs in existing file systems. FiSC [162] and eXplode [160] use model checking to find consistency bugs. Eleven82 [82] is a bug-finding tool for "recoverability bugs," where a system can crash in such a way that even after recovery, the file system is left in a state not reachable by any crash-free execution. Yggdrasil is complementary to these tools: Eleven82's automata-based bug detection allows it to explore complex optimizations, while Yggdrasil provides proofs not only of crash safety but of functional correctness.

3.9 CONCLUSION

Yggdrasil presents a new approach for building file systems with the aid of push-button verification. It guarantees correctness through a definition of file system crash refinement that is amenable to efficient SMT solving. It introduces several techniques to scale up automated verification, including the stack of abstractions and the separation of data representations. We believe that this is a promising direction since it provides a strong correctness guarantee with a low proof burden. All of Yggdrasil's source code is publicly available at `http://unsat.cs.washington.edu/projects/yggdrasil/`.

4 HYPERKERNEL: PUSH-BUTTON VERIFICATION OF AN OS KERNEL

This chapter 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.

4.1 OVERVIEW

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

As shown in Figure 8, 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.

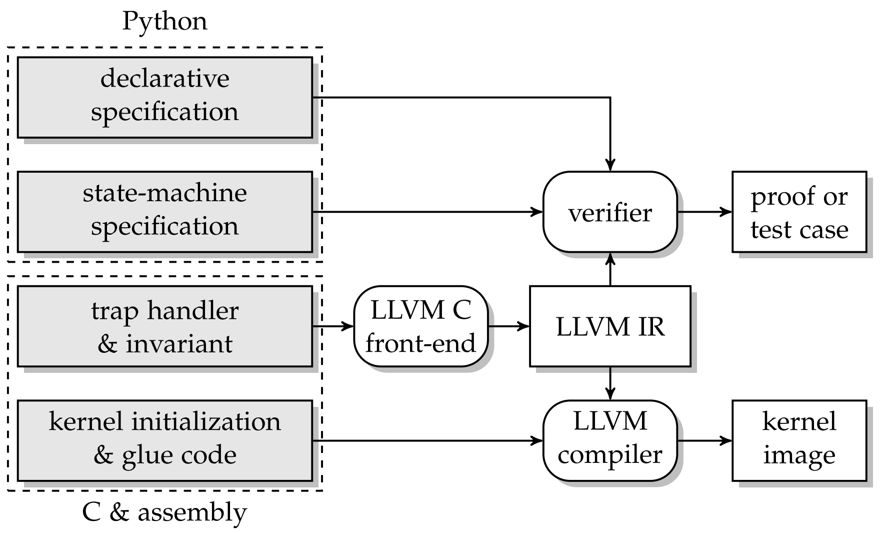


Figure 8: The Hyperkernel development flow. Rectangular boxes denote source, intermediate, and output files; rounded boxes denote compilers and verifiers. Shaded boxes denote files written by programmers.

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 Section 4.2.

4.1.1 *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 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 9 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.

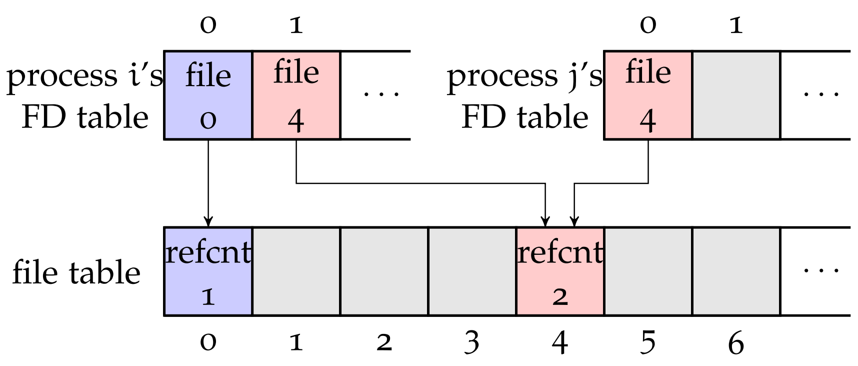


Figure 9: Per-process file descriptor (FD) tables and the system-wide file table.

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" [175]. 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 [28], 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.

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.

4.1.2 *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 [110], 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 Section 4.2.3). 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 [129] 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:

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

We provide a library to simplify the task of expressing such reference counting properties, further explained in Section 4.2.3.

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. Section 4.5.1 describes one such bug.

4.1.3 *Implementation*

The C implementation of `dup(oldfd, newfd)` in Hyperkernel closely resembles that in xv6 [32] and Unix V6 [97]. 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 [79], 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 Section 4.2 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.

4.1.4 *Verification*

The verification of Hyperkernel proves two main theorems:

Theorem 8 (Refinement)**.** The kernel implementation is a refinement of the state-machine specification.

Theorem 9 (Crosscutting)**.** The state-machine specification satisfies the declarative specification.

Proving Theorem 8 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 Section 4.1.2. Other pairs of variables are similarly constrained.

Using the equivalence function, the verifier proves Theorem 8 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 Section 4.2): 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 9, 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 8 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 9 guarantees that the state-machine specification is correct with respect to the declarative specification. For example, the declarative specification in Section 4.1.2 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. Section 4.4 will introduce separate checkers for the unverified parts of the implementation.

4.1.5 *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.

4.2 THE VERIFIER

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

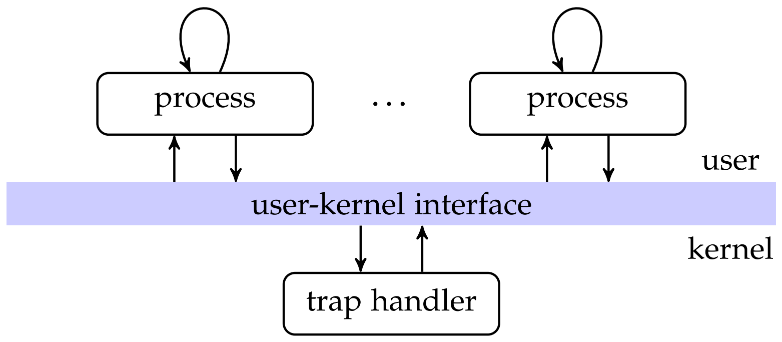


Figure 10: State transitions in Hyperkernel. At each step process execution may either stay in user space or trap into the kernel due to system calls, exceptions, or interrupts. Each trap handler in the kernel runs to completion with interrupt disabled.

verification scalable, the verifier restricts its 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 (Section 4.2.1), 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 (Section 4.2.2). 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 (Section 4.2.3).

4.2.1 *Modeling kernel behavior*

The verifier follows a standard way of modeling a kernel's execution as a state machine [77]. As shown in Figure 10, 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 [67] and AMD's Device Exclusion Vector [2] configured at boot time. In addition, the kernel conservatively considers DMA pages *volatile* (see Section 4.2.2), 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 (Section 4.1.3). 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 (Section 4.1.4). 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 8), 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 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 (Section 4.1.4).

Proving crosscutting properties (Theorem 9) 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 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 Section 4.4). Specifically, for Theorem 8, it assumes the initial state of the implementation satisfies the representation invariant I; for Theorem 9, it assumes the initial state of the state-machine specification satisfies the predicate P.

4.2.2 *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 [21]). 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 [151]: 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 [99]. 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. [92]). 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 [110]. 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.

4.2.3 *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:

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

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 [110] by observing that the *own* function must be injective due to exclusive ownership. In particular, there exists an inverse function *owned-by* such that:

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

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 Section 4.1.2) 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:

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

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 $\mathit{refcnt}(r)$, there must be a way to permute the elements of O such that exactly the first $\mathit{refcnt}(r)$ 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

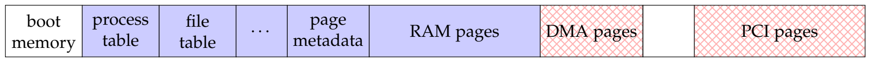


Figure 11: Memory layout in Hyperkernel: boot memory is used only during kernel initialization; shaded regions are accessible only by the CPU; and crosshatched regions are accessible by both the CPU and I/O devices.

objects of O so that only the first $\mathit{refcnt}(r)$ objects refer to the resource r:

$$\begin{aligned}&\forall r.\ \forall 0 \leqslant i < |O|.\\ &\quad \mathit{own}(\pi(r,i)) = r \Rightarrow i < \mathit{refcnt}(r)\ \wedge\\ &\quad \mathit{own}(\pi(r,i)) \neq r \Rightarrow i \geqslant \mathit{refcnt}(r)\end{aligned}$$

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

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

A library hides these details from programmers (see Section 4.1.2).

4.3 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 (Section 4.3.1), followed by common patterns of finitizing the kernel interface (Section 4.3.2). The interface allows us to implement user-space libraries to support file systems and networking (Section 4.3.3). We end this section with a discussion of limitations (Section 4.3.4).

4.3.1 *Design overview*

To make the kernel interface finite, Hyperkernel combines OS design ideas from three main sources—Dune [11], exokernels [41, 75], and seL4 [78, 79]—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 [11]. 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 [79, 81]), 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 [11, 150], 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 [41], 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 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 [40]. 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 11 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 (Section 4.2.1); 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.

4.3.2 *Designing finite interfaces*

We have designed Hyperkernel's system call interface following the rationale in Section 4.3.1. 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 Section 4.2.3 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 [75]. 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 Section 4.2.3:

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 [9], 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). Section 4.5.1 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.

4.3.3 *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 [5, 23, 142] in the future.

NETWORK. We have implemented a user-space driver for the E1000 network card (through IOMMU system calls) and ported lwIP [35] to run as a dedicated network server. We have implemented a simple HTTP server and client, capable of serving a git repository hosting this chapter.

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 [11]. The current implementation is incomplete, though it can run programs such as gzip, sha1sum, Z3, and the benchmarks we use in Section 4.5.4.

4.3.4 *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 Section 4.5.4.

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 [126, 166].

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 [40]. 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.

4.4 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 8 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 (Section 4.1.2). 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 9 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.

4.5 EXPERIENCE

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

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 12: 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.

4.5.1 *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 12 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 9), 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 (Section 4.1.2) 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 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 (Section 4.3.2) 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 [129]: 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.

4.5.2 *Development effort*

Figure 13 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 [95]. 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. [74] and Reed [128].

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

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 13: Lines of code for each component.

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 [11], 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 Section 4.3, we were able to finish the verification with a low proof burden.

VERIFICATION AND DEBUGGING. As illustrated in Section 4.1, 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 (Section 4.2.3) 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.

In our experience, Z3 was usually able to generate counterexamples for implementation bugs that violated Theorem 8. However, for violations of Theorem 9 (e.g., bugs in the state-machine specification),

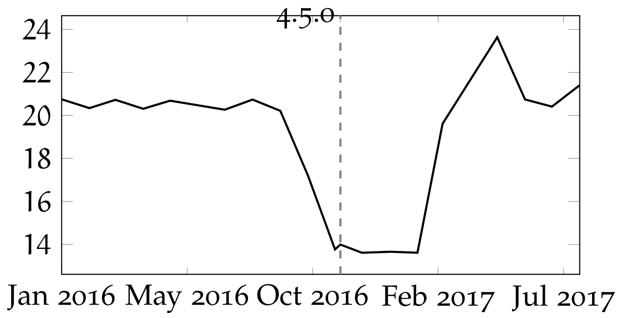


Figure 14: Verification time in minutes with Z3 commits over time. The dashed lines represent Z3 releases.

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 [68], 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 Yggdrasil, 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 (Section 4.2.3). Compared to other SMT-based verification projects like Ironclad [61] and IronFleet [62], 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 (Section 4.3.4).

4.5.3 *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 8 and 9, 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 14 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 Section 4.1.1 and the SMT encodings described in Section 4.2.3 are effective, as verification performance does not depend on the size of kernel state.

4.5.4 *Run-time performance*

To evaluate the run-time performance of Hyperkernel, we adopt benchmarks from Dune [11], excluding those that do not apply to Hyperkernel (e.g., ptrace). Figure 15 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 (Section 4.3.3), 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 Section 4.5.5. 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 [6], 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 [11]: for the worst-case scenarios, the use of hypercalls incurs a 5× overhead compared to `syscall`; on the other hand, de-

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

pending 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 (Section 4.3.3) is a placeholder for demonstrating the usability of the kernel interface. We hope to incorporate high-performance stacks [12, 121] in the future.

4.5.5 *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 16: 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 [3]. For non-x86 architectures like ARM, the system call and hypercall instructions have similar performance [33], and so exploring Hyperkernel on ARM would be attractive future work [72].

4.6 RELATED WORK

VERIFIED OS KERNELS. OS kernel verification has long been a research objective. Early efforts in this direction include UCLA Secure Unix [154], PSOS [44], and KIT [14]; see Klein et al. [79] 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 [78, 79]. Hyperkernel's design is inspired in several places by seL4, as discussed in Section 4.3.1. In contrast to seL4, however, we aimed to keep Hyperkernel's design as close to a classic Unix-like kernel as possible, while enabling au-

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

tomated verification with SMT solvers. In support of this goal, we made the Hyperkernel interface finite, avoiding unbounded loops, recursion, or complex data structures.

Ironclad establishes end-to-end security properties from the application layer down to kernel assembly [61]. Ironclad builds on the Verve type-safe kernel [159], and uses the Dafny verifier [93], which is built on Z3, to help automate proofs. Similarly, ExpressOS [101] verifies security properties of a kernel using Dafny. As discussed in Section 4.5.2, 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 [55] and Xu et al.'s framework for reasoning about interrupts in the μC/OS-II kernel [158]. Both projects use the Coq interactive theorem prover [149] 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 Section 4.5.2, Hyperkernel builds on the lessons learned from Yggdrasil. 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 (Section 4.2.3). 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 [131], 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 [11], in which each process is virtualized and so has more direct control over hardware features such as interrupt vectors [11]. 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 [41], which offers low-level hardware access to applications for extensibility. As discussed in Section 4.3.1, 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 [172] formalizes LLVM IR in Coq; Alive [99] formalizes LLVM IR to verify the correctness of compiler optimizations; SMACK [127] translates LLVM IR to the Boogie verification language [8]; KLEE [21] and UFO [4] translate LLVM IR to SMT for verification; and Vigor uses a modified KLEE as part of its toolchain to verify a network address translator [164].

The Hyperkernel verifier also translates LLVM IR to SMT to prove Theorem 8 (Section 4.2.2). 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 [151]. The verifier also uses a simple memory model tailored for kernel verification, translating memory accesses to uninterpreted functions.

4.7 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://unsat.cs.washington.edu/projects/hyperkernel/`.

5 NICKEL: A FRAMEWORK FOR DESIGN AND VERIFICATION OF INFORMATION FLOW CONTROL SYSTEMS

Nickel is a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface, which can be exploited to circumvent the enforcement of information flow policies. Nickel provides a formulation of noninterference amenable to automated verification, allowing developers to specify an intended policy of permitted information flows. It invokes the Z3 SMT solver to verify that both an interface specification and an implementation satisfy noninterference with respect to the policy; if verification fails, it generates counterexamples to illustrate covert channels that cause the violation.

Using Nickel, we have designed, implemented, and verified NiStar, the first OS kernel for decentralized information flow control that provides (1) a precise specification for its interface, (2) a formal proof that the interface specification is free of covert channels, and (3) a formal proof that the implementation preserves noninterference. We have also applied Nickel to verify isolation in a small OS kernel, NiKOS, and reproduce known covert channels in the ARINC 653 avionics standard. Our experience shows that Nickel is effective in identifying and ruling out covert channels, and that it can verify noninterference for systems with a low proof burden.

5.1 COVERT CHANNELS IN INTERFACES

Nickel's main goal is to help developers identify and eliminate covert channels in the interface of an information flow control system. This section surveys common examples of covert channels and shows how to apply noninterference to understand them.

Consider two threads T_1 and T_2 that are prohibited from communicating as per the information flow policy. What kinds of interface operations can be exploited by the two threads to collude and bypass the policy (or equivalently, allow an adversarial T_2 to infer secret information from an uncooperative T_1)? As a simple example, if an operation introduces shared memory locations that both threads can read and write, then the two threads can use these memory locations as covert channels to transfer information. Unintended covert channels, however, are often subtle and difficult to spot, as detailed next.

RESOURCE NAMES. Resource names, such as thread identifiers, page numbers, and port numbers, can be used to encode information. Consider a system call `spawn` that creates new threads with sequential identifiers. Thread T_2 first spawns a thread with an identifier, say, 3; the other thread T_1 then spawns x times, creating threads with identifiers from 4 to $x + 3$; and thread T_2 spawns another thread, whose identifier will be $x + 3 + 1$. In doing so, thread T_2 learns the secret x from T_1 through the difference of the identifiers of the two threads it has created [31, §5].

RESOURCE EXHAUSTION. Suppose that the system has a total of N pages shared by all threads. Thread T_1 first allocates $N - 1$ pages, and encodes a one-bit secret by either allocating the last page or not. The other thread T_2 then tries to allocate one page and learns the secret based on whether the allocation succeeds [169, §3]. This covert channel is effective especially when a resource is limited and can be easily exhausted.

STATISTICAL INFORMATION. A thread's world-readable information, such as its name, number of open file descriptors, and CPU and memory usage, can be used to encode secret data or by adversarial threads to learn secrets [69, 171]. For example, if thread T_1's memory usage is accessible to another thread T_2 through `procfs` or system calls, T_1 could leak a secret x by allocating x pages.

ERROR HANDLING. Error handling is known to be prone to information leakage [20], such as the TENEX password-guessing attack using page faults [90] and the POODLE attack against TLS [108]. As an example, consider a system call for querying the status of a page, which returns `-ENOENT` if the given page is free and `-EACCES` if the page is in use but not accessible by the calling thread. Thread T_1 encodes a one-bit secret by allocating a particular page or not; the other thread T_2 queries the status of that page and learns the secret based on whether the error code is `-ENOENT` or `-EACCES`.

SCHEDULING. Suppose an OS kernel uses a round-robin scheduler that distributes time slices evenly among threads. Thread T_1 encodes a secret by forking a number of threads (e.g., a fork bomb), which causes the other thread T_2 to observe the reduction of time allocated for itself and learn the secret from T_1; alternatively, T_2 can continuously ping a remote server, which will learn the secret from the time between pings [169, §9]. Access to only logical time suffices for such covert channels.

EXTERNAL DEVICES AND SERVICES. Suppose the system allows threads to communicate with external devices and services. Thread

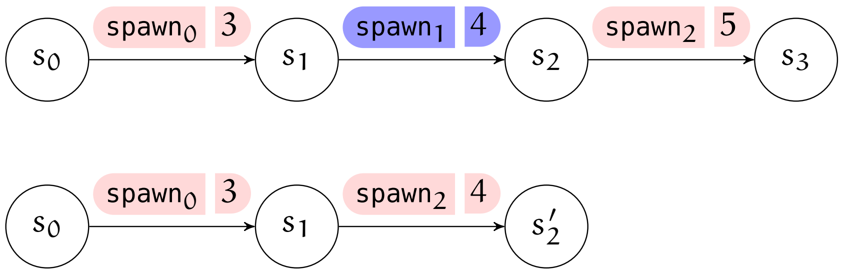


Figure 17: The output of `spawn`$_2$ changes from 5 in the original trace (first row) to 4 in the purged trace (second row), indicating a covert channel. Circles denote states, arrows denote state transitions, left half-circles denote actions, and right half-circles denote outputs.

T_1 can write secret data to the registers of a device, or encode the secret as the frequency of accessing a device or even through a service bill [89]; the other thread T_2 can then retrieve the secret at a later time from the same device or service.

mutable labels. Many information flow control systems express security policies by assigning *labels* to objects. Label changes complicate such systems and can lead to covert channels [34]. As an example, consider a system where each thread can be labeled as either tainted or untainted. The system enforces a tainting policy: a tainted thread cannot transfer information to an untainted thread without tainting it. To enforce this policy, the system raises the label of an untainted thread to tainted when another tainted thread sends data to it. Suppose thread T_1 is tainted and thread T_2 is untainted. To bypass the policy, T_2 first spawns an untainted helper thread H. T_1 encodes a one-bit secret by choosing whether to send data to taint H, which in turn chooses to send data to T_2 *only* if it is untainted and do nothing otherwise. In this way, T_2 learns the secret from T_1 by whether it receives data from H, without becoming tainted itself [83, §3].

5.1.1 *Applying noninterference*

Given two threads T_1 and T_2 that are prohibited from communicating with each other, noninterference states that the output of operations in one thread should not be affected by whether operations in the other thread occur. Now we will show how to apply noninterference to uncover covert channels.

Take the `spawn` system call as an example, which returns sequential thread identifiers and introduces a covert channel due to resource names. Figure 17 illustrates this channel. We denote an action of invoking a system call as a left half-circle `spawn` and its return value as a right half-circle 3. We use different colors to distinguish system calls from different threads: `spawn`$_1$ in T_1; `spawn`$_0$ and `spawn`$_2$ in T_2.

We apply noninterference to uncover the covert channel introduced by `spawn` in three steps. First, construct a trace of actions from both threads, for instance, `spawn`$_0$ `spawn`$_1$ `spawn`$_2$. Assume that the corresponding return values (i.e., outputs) are 3 4 5, as `spawn` sequentially allocates identifiers. Second, to examine possible effects of T_1 on T_2, construct a new trace that *purges* the actions from T_1 and retains the actions only from T_2, resulting in `spawn`$_0$ `spawn`$_2$. Third, replay this purged trace to the system, obtaining a new sequence of outputs 3 4. This sequence differs from the original output of the same actions, which is 3 5. The change of output in T_2 (in particular, the return value of `spawn`$_2$) caused by an action in T_1 violates noninterference, indicating a covert channel with which T_1 may transfer information to T_2. On the other hand, with a version of `spawn` that does not introduce a covert channel, the outputs of T_2's actions in the purged and original traces would be the same.

One can similarly apply noninterference to uncover the other covert channels described in this section. The challenge is to find a trace of actions that manifests the covert channel, and if there are no such channels, to exhaustively show that no trace violates noninterference. Nickel automates this task using formal verification techniques, as we will describe next.

5.2 PROVING NONINTERFERENCE

This section formalizes the notion of noninterference used in Nickel and presents the main theorems that enable Nickel to prove noninterference for systems.

First, we address how to specify the intended policy of an information flow control system. The policy is trusted as the top-level specification of the system, which will be used to catch and fix potential covert channels in both the interface specification and the implementation (Section 5.2.1).

Next, we give a formal definition of noninterference in terms of traces of actions, which precisely captures whether an interface specification satisfies a given policy (Section 5.2.2).

To prove noninterference for an interface specification, Nickel introduces an unwinding verification strategy that requires reasoning only about individual actions, rather than traces of actions (Section 5.2.3). To extend the guarantee of noninterference to an implementation, Nickel introduces a restricted form of refinement that preserves noninterference (Section 5.2.4). Both strategies are amenable to automated verification using an SMT solver.

We end this section with a discussion of the limitations of the Nickel approach (Section 5.2.5).

5.2.1 *Policy*

We model the execution of a system as a state machine in a standard way [138]. A system $\mathcal{M}$ is defined as a tuple $\langle A, O, S, \texttt{init}, \texttt{step}, \texttt{output}\rangle$, where A is the set of actions, O is the set of output values, S is the set of states, `init` is the initial state, $\texttt{step} : S \times A \to S$ is the state-transition function, and $\texttt{output} : S \times A \to O$ is the output function.

An action transitions the system from state to state. In the context of an OS, an action can be either a user-space operation (e.g., memory access), or the handling of a trap due to system calls, exceptions, or scheduling. Each action consists of an operation identifier (e.g., the system call number) and arguments. We write $\texttt{output}(s, a)$ and $\texttt{step}(s, a)$ to denote the output value (e.g., the return value of a system call) and the next state, respectively, for the state s and action a. Actions are considered to be atomic; for instance, we assume that an OS kernel executes each trap handler with interrupts disabled on a uniprocessor system [78].

A *trace* is a sequence of actions. We use $\texttt{run}(s, tr)$ to denote the state produced by executing each action in trace tr starting from state s. The `run` function is defined as follows:

$$\begin{aligned} \texttt{run}(s, \epsilon) &\coloneqq s \\ \texttt{run}(s, a \circ tr) &\coloneqq \texttt{run}(\texttt{step}(s, a), tr). \end{aligned}$$

Here, ϵ denotes the empty trace, and $a \circ tr$ denotes the concatenation of action a and trace tr.

Definition 3 (Information Flow Policy)**.** A policy $\mathcal{P}$ for system $\mathcal{M}$ is defined as a tuple $\langle D, \rightsquigarrow, R\rangle$, where D is the set of *domains*, $\rightsquigarrow \subseteq (D \times D)$ is the can-flow-to relation between two domains, and the function $R : A \times S \to D$ maps an action with a state to a domain.

Intuitively, a domain is an abstract representation of the exercised authority of an action. A policy associates each action a performed from state s with a domain, denoted by $R(a, s)$; the can-flow-to relation $\rightsquigarrow$ defines permitted information flows among these domains. The goal of a policy is to explicitly specify permitted flows and ensure that any trace of actions, given their specifications, will *not* lead to covert channels that enable unintended flows and violate the policy.

Below we show the policies for two example systems. We write $u \rightsquigarrow v$ and $u \not\rightsquigarrow v$ to mean $(u, v) \in \rightsquigarrow$ and $(u, v) \notin \rightsquigarrow$, respectively.

Example (Tainting)**.** Consider the label-based system mentioned in Section 5.1: it has a number of threads, where the label of each thread is either tainted or untainted. The system enforces a tainting policy as depicted in Figure 18. The policy permits information flow from untainted threads to either untainted or tainted threads, and between

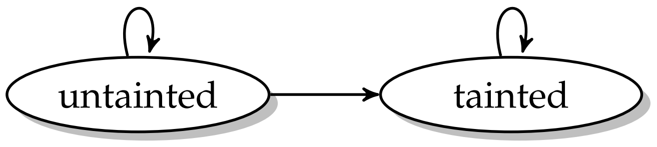


Figure 18: The tainting policy: information cannot flow from tainted threads to untainted threads.

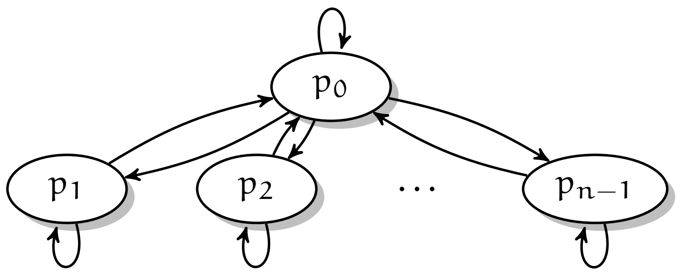


Figure 19: The isolation policy of NiKOS: information cannot flow between any two of the regular processes $p_1, p_2, \ldots, p_{n-1}$ (except through the scheduler p_0 indirectly).

two tainted threads, but it prohibits untainted threads from directly communicating with tainted ones.

For this policy, we designate $\{tainted, untainted\}$ as the set of domains. The can-flow-to relation consists of the following three permitted flows: *tainted* $\rightsquigarrow$ *tainted*, *untainted* $\rightsquigarrow$ *untainted*, and *untainted* $\rightsquigarrow$ *tainted*. The *R* function returns the label of the thread currently running. NiStar employs a more sophisticated version of this policy using DIFC (see Section 5.5).

Example (Isolation)**.** Consider a Unix-like kernel with n processes: a special scheduler process p_0, and regular processes $p_1, p_2, \ldots, p_{n-1}$. The system enforces a process isolation policy as depicted in Figure 19, which permits information flows from a process to itself, from the scheduler to any process, and from any process to the scheduler; no information flow is permitted between any two regular processes except indirectly through the scheduler [31].

To specify this isolation policy, we designate the processes $\{p_0, p_1, \ldots, p_{n-1}\}$ as the set of domains, where p_0 is the scheduler. The can-flow-to relation consists of the permitted flows $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$, for all $i \in [0, n-1]$. The *R* function returns the currently running process as the domain for system call actions, and returns the scheduler p_0 as the domain for context switching actions. NiKOS employs this policy (see Section 5.6).

We highlight two features in our policy definition (3). First, it allows the can-flow-to relation $\rightsquigarrow$ to be *intransitive* [138]. For instance, the isolation policy permits processes p_1 and p_2 to communicate through the scheduler, but prohibits them from communicating directly with

$$\mathtt{sources}(\epsilon, u, s) := \{u\}$$

$$\mathtt{sources}(a \circ tr, u, s) := \mathtt{sources}(tr, u, \mathtt{step}(s, a)) \cup \begin{cases} \{R(a, s)\} & \text{if } \exists v \in \mathtt{sources}(tr, u, \mathtt{step}(s, a)).\ R(a, s) \rightsquigarrow v \\ \varnothing & \text{otherwise.} \end{cases}$$

Figure 20: $\mathtt{sources}(tr, u, s)$ is the set of domains that are allowed to influence domain u over a trace tr, starting from state s.

$$\mathtt{purge}(\epsilon, u, s) := \{\epsilon\}$$

$$\mathtt{purge}(a \circ tr, u, s) := \{a \circ tr' \mid tr' \in \mathtt{purge}(tr, u, \mathtt{step}(s, a))\} \cup \begin{cases} \varnothing & \text{if } R(a, s) \in \mathtt{sources}(a \circ tr, u, s) \\ \mathtt{purge}(tr, u, s) & \text{otherwise.} \end{cases}$$

Figure 21: $\mathtt{purge}(tr, u, s)$ is the set of all sub-traces of tr that retain the actions that are allowed to influence domain u, starting from state s.

each other. In other words, $p_1 \rightsquigarrow p_0$ and $p_0 \rightsquigarrow p_2$ do *not* have to imply $p_1 \rightsquigarrow p_2$, though that would also be accepted by Nickel if it were the intended policy.

This generality enables Nickel to support a broad range of policies, as practical systems often need *downgrading* operations (e.g., intentional declassification and endorsement) [96]. As a simple example, a system may prefer to have an untrusted application send data to an encryption program, which in turn is permitted to reach the network, while the application itself is prohibited from sending data directly over the network. Such policies require intransitive can-flow-to relations [138, 165].

Second, in classical noninterference [52, 138], the R function is state-independent ($A \to D$). The definition of R used in Nickel is *state-dependent* ($A \times S \to D$). This extension is necessary for reasoning about many systems in which the domain (i.e., authority) of an action depends on the currently running thread or process [112, 140]. As we will show next, we have developed a definition of noninterference and theorems for proving noninterference that accommodate this extension.

5.2.2 *Noninterference*

Given a system and a policy for the system, what kind of action can violate the policy and introduce covert channels? As described in Section 5.1, to check for noninterference, one can construct a trace of actions, obtain a purged trace by removing actions from the original trace as per the policy, and compare the output of the corresponding actions in both traces—any change of output indicates a covert channel. Below we give a precise definition of noninterference that captures this intuition, in three steps.

First, suppose that a system has executed a trace tr to reach the state $\hat{s} = \texttt{run}(\texttt{init}, tr)$, and is about to perform action $\hat{a}$ next. To construct a purged trace of tr, we need to identify the actions that the policy permits to influence a domain u and therefore should be retained in the trace. This set is defined using the $\texttt{sources}(tr, u, s)$ function shown in Figure 20, which returns the set of domains that can transfer information to domain u over trace tr from state s, either directly specified by the can-flow-to relation or indirectly through the domain of another intermediate action in the trace.

Second, to obtain a purged trace that retains the actions identified by $\texttt{sources}$, we define the $\texttt{purge}(tr, u, s)$ function as shown in Figure 21. It returns the set of all sub-traces of tr where each action in the sources of u from state s has been retained; the actions whose domains are not identified by $\texttt{sources}$ are optionally removed.

Third, let tr' denote a purged trace in the set $\texttt{purge}(tr, R(\hat{a}, \hat{s}), \texttt{init})$; like other traces in this set, tr' is obtained by retaining actions in trace tr that can transfer information to action $\hat{a}$. Now let's replay the purged trace tr' from the start, resulting in a new state $\hat{s}' = \texttt{run}(\texttt{init}, tr')$. If the system satisfies noninterference for the policy, then invoking $\hat{a}$ from state $\hat{s}$ should produce the same output as invoking $\hat{a}$ from state $\hat{s}'$.

Formally, we define noninterference as follows:

Definition 4 (Noninterference)**.** Given a system $\mathcal{M} = \langle A, O, S, \texttt{init}, \texttt{step}, \texttt{output} \rangle$ and a policy $\mathcal{P} = \langle D, \rightsquigarrow, R \rangle$, $\mathcal{M}$ satisfies noninterference for $\mathcal{P}$ if and only if the following holds for any trace tr, action a, and purged trace $tr' \in \texttt{purge}(tr, R(a, \texttt{run}(\texttt{init}, tr)), \texttt{init})$:

$$\texttt{output}(\texttt{run}(\texttt{init}, tr), a) = \texttt{output}(\texttt{run}(\texttt{init}, tr'), a).$$

To ensure that our definition of noninterference is reasonable, we show two properties of this definition. First, recall that we use a state-dependent R function; if R is restricted to be state-independent, that is, $R(a, s) = R(a)$ holds for any a and s, then our definition reduces to classical noninterference [138], suggesting that our definition is a natural extension.

Second, a reasonable definition of noninterference should be *monotonic* [39]: a system satisfying noninterference for some policy should also satisfy noninterference for a more relaxed policy in which more flows are permitted. More formally, given two policies $\mathcal{P} = \langle D, \rightsquigarrow, R \rangle$ and $\mathcal{P}' = \langle D, \rightsquigarrow', R \rangle$, we say $\mathcal{P}'$ *contains* $\mathcal{P}$ to mean that any flow permitted by $\mathcal{P}$ is also permitted by $\mathcal{P}'$ (i.e., $\rightsquigarrow \subseteq \rightsquigarrow'$). We have proved the following monotonicity property as a sanity check on our definition of noninterference: if a system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$, then it also satisfies noninterference for any policy $\mathcal{P}'$ that contains $\mathcal{P}$.

$\mathcal{I}$ is a state invariant:

$$\mathcal{I}(\mathtt{init}) \wedge (\mathcal{I}(s) \Rightarrow \mathcal{I}(\mathtt{step}(s,a)))$$

$\overset{u}{\approx}$ is an equivalence relation:

$\overset{u}{\approx}$ is reflexive, symmetric, and transitive

$\overset{u}{\approx}$ is consistent with R:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{R(a,s)}{\approx} t \Rightarrow R(a,s) = R(a,t)$$

$\overset{u}{\approx}$ is consistent with $\rightsquigarrow$:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \Rightarrow (R(a,s) \rightsquigarrow u \iff R(a,t) \rightsquigarrow u)$$

output consistency:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{R(a,s)}{\approx} t \Rightarrow \mathtt{output}(s,a) = \mathtt{output}(t,a)$$

local respect:

$$\mathcal{I}(s) \wedge R(a,s) \not\rightsquigarrow u \Rightarrow s \overset{u}{\approx} \mathtt{step}(s,a)$$

weak step consistency:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge s \overset{R(a,s)}{\approx} t \Rightarrow \mathtt{step}(s,a) \overset{u}{\approx} \mathtt{step}(t,a)$$

Figure 22: Unwinding conditions. Each formula is universally quantified over its free variables, such as domain u, action a, and states s and t.

5.2.3 *Unwinding*

It is difficult to directly apply 4 to prove noninterference for a given system and policy, as it requires reasoning about all possible traces. A standard approach is to define a set of *unwinding conditions*, which together imply noninterference but require reasoning only about individual actions. We generalize the classical unwinding conditions given by Rushby [138] to obtain an unwinding theorem that accommodates our state-dependent `dom` function and is amenable to automated verification. Proving noninterference using the unwinding theorem requires two extra inputs from developers: a *state invariant* and an *observational equivalence* relation, as described next.

A state invariant $\mathcal{I}$ [87] is a state predicate that must hold on all *reachable* states (i.e., the set of states produced by running any trace starting from the `init` state). The state invariant overapproximates the set of reachable states, as it may also hold for unreachable states. If the unwinding theorem holds for states satisfying $\mathcal{I}$, then it holds for all reachable states of the system. We use this overapproximation to enable automation: in contrast to reachability, which cannot be expressed in first-order logic, the state invariant can be both expressed and effectively checked with an SMT solver.

The next input required for the unwinding theorem is an observational equivalence relation $\approx \subseteq (D \times S \times S)$. The observational equivalence describes, for each domain, the set of states that appear to that domain to be indistinguishable. We write $\overset{u}{\approx}$ to mean the binary relation $\{(s, t) \mid (u, s, t) \in \approx\}$ relating all equivalent states for domain u, and $s \overset{u}{\approx} t$ to mean $(u, s, t) \in \approx$.

We then define the unwinding conditions of system $\mathcal{M}$ for policy $\mathcal{P}$, shown in Figure 22, and prove the following unwinding theorem:

Theorem 10 (Unwinding)**.** A system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$ if there exists a state invariant $\mathcal{I}$ and an observational equivalence relation $\approx$ for which the unwinding conditions in Figure 22 hold.

The unwinding theorem obviates the need to reason about traces to prove noninterference; instead, it suffices to show that the unwinding conditions hold for each action. This theorem enables Nickel to automate the checking using the Z3 SMT solver (see Section 5.3). Both the state invariant $\mathcal{I}$ and the observational equivalence relation $\approx$ are *untrusted*: any instances that satisfy the conditions are sufficient to establish noninterference.

We give some intuition behind the unwinding theorem. The first four conditions are natural: they ask for a reasonable state variant $\mathcal{I}$ and observational equivalence relation $\approx$ (i.e., $\overset{u}{\approx}$ should be an equivalence relation and be consistent with the policy). The remaining three conditions, *output consistency*, *local respect*, and *weak step consistency*, provide more hints to interface design, as follows. As a shorthand, we say "objects" to mean individual storage locations in the system state.

First, the output of an action should depend only on objects that the domain of the action can read. Restricting the output prevents an adversarial application from inferring information about system state via return values, such as the error-handling channel described in Section 5.1.

Second, if an action attempts to modify an object, the domain of the action should be able to write to that object, and its new value should depend only on the old value and objects that the domain of the action can read. This requirement prevents unintended flows while updating the system state, such as the resource-name channel introduced by `spawn` sequentially allocating identifiers.

Third, if an action attempts to create a new object, that new object should have equal or less authority than the domain of the action; similarly, if an object becomes newly readable after an action, then the domain of the action should have been able to read that object before the call. These restrictions preclude "runaway" authority—no action can arbitrarily increase the authority of its domain, or create an object more powerful than itself.

5.2.4 *Refinement*

Refinement is widely used for verifying systems: developers describe the intended system behavior as a high level, abstract specification and check that any behavior exhibited by a low level, concrete implementation is allowed by the specification. Refinement allows developers to reason about many properties of the system at the specification level, which is often simpler than reasoning about the implementation directly.

In our case, it would be ideal to prove noninterference (using the unwinding theorem) for an interface specification, and extend that guarantee to an implementation that refines the specification. However, it is well known that noninterference is generally *not* preserved under refinement [53, 102]; for example, the implementation may introduce extra stuttering steps that leak information. Nickel supports a restricted form of refinement over state machines and policies. We show here that this refinement preserves noninterference as defined in Section 5.2.2.

Let's consider the following systems:

- $\mathcal{M}_1 = \langle A, O, S_1, \mathtt{init}_1, \mathtt{step}_1, \mathtt{output}_1 \rangle$, and
- $\mathcal{M}_2 = \langle A, O, S_2, \mathtt{init}_2, \mathtt{step}_2, \mathtt{output}_2 \rangle$.

These two systems share the set of actions A and the set of outputs O, but differ in the state spaces, as well as the state-transition and output functions. One may consider $\mathcal{M}_1$ as the specification and $\mathcal{M}_2$ as the implementation. We say that $\mathcal{M}_2$ is a *data refinement* of $\mathcal{M}_1$ to mean that they produce the same output for any trace [65, 87]. Data refinement is particularly useful for verifying systems with a well-defined interface, such as OS kernels [77].

A standard way to prove data refinement of $\mathcal{M}_1$ by $\mathcal{M}_2$ is to ask developers to identify a data refinement relation $\propto \subseteq (S_2 \times S_1)$; we write $s_2 \propto s_1$ to mean $(s_2, s_1) \in \propto$. Let $\mathcal{I}_2$ denote a state invariant for $\mathcal{M}_2$. To prove that $\mathcal{M}_2$ is a data refinement of $\mathcal{M}_1$, it suffices to show that the following *refinement conditions* hold:

- $\mathtt{init}_2 \propto \mathtt{init}_1$.
- $\mathcal{I}_2(s_2) \land s_2 \propto s_1 \Rightarrow \mathtt{step}_2(s_2, a) \propto \mathtt{step}_1(s_1, a)$.
- $\mathcal{I}_2(s_2) \land s_2 \propto s_1 \Rightarrow \mathtt{output}_2(s_2, a) = \mathtt{output}_1(s_1, a)$.

Each formula is universally quantified over s_1, s_2, and a.

Given policies $\mathcal{P}_1 = \langle D, \rightsquigarrow, R_1 \rangle$ and $\mathcal{P}_2 = \langle D, \rightsquigarrow, R_2 \rangle$ for systems $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, we say that $\mathcal{P}_2$ is a *policy refinement* of $\mathcal{P}_1$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if and only if the following holds for any action a and trace *tr*: $R_1(a, \mathtt{run}_1(\mathtt{init}_1, tr)) = R_2(a, \mathtt{run}_2(\mathtt{init}_2, tr))$. Here $\mathtt{run}_1$ and $\mathtt{run}_2$ apply a trace starting from a given state for $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively (Section 5.2.2).

With these notions of data refinement and policy refinement, we have proved the following refinement theorem for noninterference:

Theorem 11 (Refinement)**.** Given two systems $\mathcal{M}_1$ and $\mathcal{M}_2$ and policy $\mathcal{P}$ for $\mathcal{M}_1$, $\mathcal{M}_2$ satisfies noninterference for any policy refinement of $\mathcal{P}$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if:

- there exists a state invariant $\mathcal{I}_1$ of system $\mathcal{M}_1$ and an observational equivalence relation $\approx$ for which the unwinding conditions of $\mathcal{M}_1$ for $\mathcal{P}$ hold; and
- there exists a state invariant $\mathcal{I}_2$ of system $\mathcal{M}_2$ and a data refinement relation $\propto$ for which the refinement conditions of $\mathcal{M}_1$ by $\mathcal{M}_2$ hold.

The refinement theorem enables Nickel to check noninterference for an implementation by checking the unwinding conditions for the interface specification and the refinement conditions (see Section 5.3). As with the unwinding theorem, the state invariants $\mathcal{I}_1$ and $\mathcal{I}_2$, the observational equivalence relation $\approx$, and the data refinement relation $\propto$ are *untrusted* for establishing noninterference.

5.2.5 *Discussion and limitations*

Nickel's formulation of noninterference falls into the category of *intransitive noninterference* [138]; in other words, it allows the can-flow-to relation of a policy to be either transitive or intransitive. As explained in Section 5.2.1, this flexibility is particularly useful for verifying practical systems, which often require downgrading operations. In addition, unlike classical noninterference, Nickel uses a state-dependent R function, inspired by the formulation used to verify multiapplicative smart cards [140] and the seL4 kernel [113].

Nickel extends previous work in the following ways: the formulation supports a general set of policies and systems, which enables us to verify DIFC in NiStar (Section 5.5) and isolation in NiKOS and ARINC 653 (Section 5.6); all of its verification conditions for unwinding and refinement are expressible using an SMT solver, enabling automated verification to minimize the proof burden; and it provides a restricted form of refinement that preserves noninterference from an interface specification to an implementation.

Nickel's formulation of noninterference has the following limitations. It cannot uncover covert channels based on resources that are not captured in the interface specification, such as timing, sound, and energy. Modeling the effects of these resources is an orthogonal problem. Recent microarchitectural attacks [19, 80, 98] suggest the need for new hardware designs and primitives in order to eliminate such channels [47, 50].

Nickel does not support reasoning about concurrent systems. Concurrency is challenging not just for verification in general, but also for its implications on noninterference [146, 153]. In addition, Nickel models systems as deterministic state machines and requires developers to eliminate nondeterminism from the interface design (see

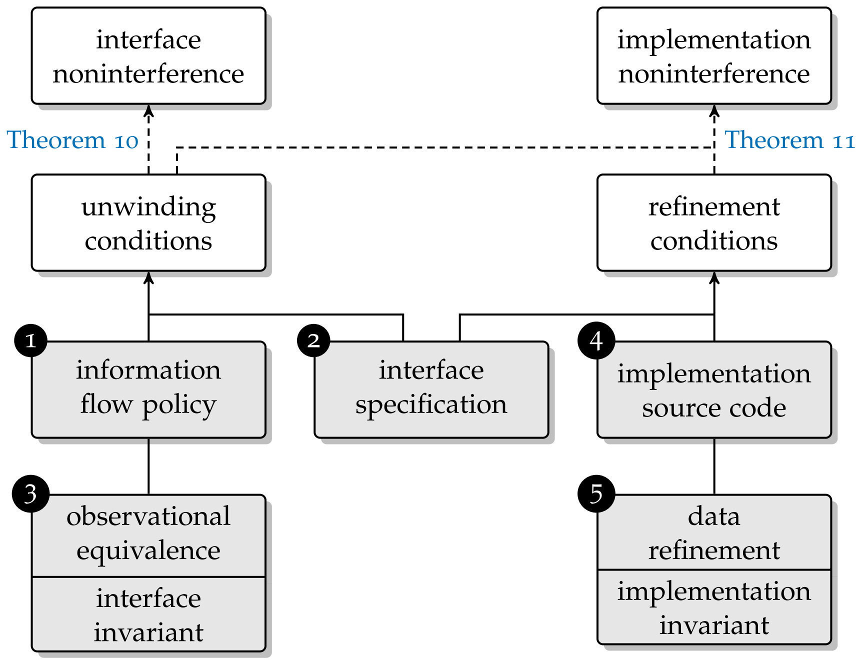


Figure 23: An overview of development flow using Nickel. Shaded boxes denote files written by system developers and the rest are provided by the framework. Circled numbers denote the steps. Solid and dashed arrows denote proof flows in SMT and Coq, respectively.

Section 5.4). This requirement enables better proof automation and simplifies noninterference under refinement, but it restricts the types of interfaces that Nickel can verify [119].

Nickel's can-flow-to relation $\rightsquigarrow$ is state-independent, which means that Nickel cannot reason about dynamic, state-dependent policies [39] (though state-dependent R functions partially compensate for this limitation). Moreover, Nickel's notion of refinement requires the interface specification and the implementation to use the same sets of actions and domains; this equality is sufficient for verifying systems like NiStar and NiKOS. Extending Nickel to support dynamic policies and more flexible refinements [105] would be useful future work.

5.3 USING NICKEL

This section explains how the Nickel framework works and describes the steps needed to design and verify information flow control systems using Nickel.

Figure 23 depicts an overview of the Nickel framework and the required inputs from system developers (shaded boxes with circled numbers). As part of the framework, the unwinding and refinement theorems (Theorem 10 and Theorem 11) serve as the metatheory for Nickel. We have formalized and proved both theorems using the Coq interactive theorem prover [149].

Developers write the system implementation in C and specify the rest of the inputs in Python. In particular, the development flow of using Nickel is the following:

1. Write the intended information flow policy to serve as the top-level specification of the system.
2. Model the system as a state machine and write a precise specification of each operation in the interface.
3. Construct a state invariant and observational equivalence for the interface specification, and invoke Nickel to check the unwinding conditions.
4. Implement each operation in the interface.
5. Construct a state invariant for the implementation and data refinement between the interface specification and the implementation, and invoke Nickel to check the refinement conditions.

Nickel extends the specification and verification infrastructure from Hyperkernel to support reasoning about noninterference. It reduces all the inputs to SMT constraints—for instance, by performing symbolic execution on the LLVM intermediate representation of the implementation—and invokes Z3 to verify noninterference by checking the unwinding and refinement conditions. As with Hyperkernel, the initialization and glue code of the implementation is unverified.

For verifying noninterference for an interface specification, the trusted computing base includes the information flow policy, the checker of unwinding conditions from Nickel, and Z3. For verifying noninterference for an implementation, it further includes the checker of refinement conditions from Nickel and the unverified initialization and glue code of the implementation.

Below we highlight two features of the development flow using Nickel.

A SIMPLE API FOR SPECIFYING THE POLICY. As described in Section 5.2.1, a policy consists of a set of domains, a can-flow-to relation over domains, and a *R* function associating each action in a state with a domain. Nickel provides a simple and intuitive API for specifying policies.

As an example, recall the isolation policy in Figure 19: each process p_i is a domain; the permitted flows in the system are: $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$ for $i \in [0, n-1]$. In Nickel, this policy is written as follows:

```
class ProcessDomain:
    def __init__(self, pid):
        self.pid = pid

    def can_flow_to(self, other):
        # Or is a built-in logical operator
        return Or(
            self.pid == 0,          # p0 ~> pi
            other.pid == 0,         # pi ~> p0
            self.pid == other.pid, # pi ~> pi
        )
```

In addition, the `dom` function of this policy returns the process currently running by default, or the scheduler p_0 for context switching actions (say, the `yield` system call):

```
class State:
    current = PidT() # PidT is an integer type
    ...

def dom(action, state):
    if action.name == 'yield':
       return ProcessDomain(0)
    else:
       return ProcessDomain(state.current)
```

This is all Nickel needs for the policy of NiKOS (Section 5.6).

Since a policy is the top-level specification of a system and must be trusted, developers should carefully audit the policy and ensure that it captures the design intention. We hope that the simple API for policies provided by Nickel makes auditing easier.

DEBUGGING THROUGH COUNTEREXAMPLES. To verify noninterference for an interface specification, Nickel checks the unwinding conditions from Theorem 10. If verification fails, Nickel produces a counterexample that illustrates the violation, including the operation name, an assignment of the operation arguments and system state(s), and the offending unwinding conditions.

Counterexamples provide useful information for debugging two types of failures. First, the violation may be in the interface specification, indicating a covert channel. Developers can use the counterexample to understand the violation and iterate on the interface design (see Section 5.4 for guidelines) until verification passes. Second, the state invariant or the observational equivalence may be insufficient to establish noninterference. Developers can consult the counterexample to fix these inputs. Debugging the verification of an implementation follows similar steps.

5.4 DESIGNING INTERFACES FOR NONINTERFERENCE

We have applied Nickel to verify noninterference in three systems: NiStar (Section 5.5), NiKOS (Section 5.6), and ARINC 653 (Section 5.6). While they have different information flow policies, our experience with these systems suggests several common guidelines for interface design.

PERFORM FLOW CHECKS EARLY. In general, operations need to validate parameters, especially those from untrusted sources (e.g., user-specified values in system calls), and return error codes indicating the cause of failure. As described in Section 5.1, returning error codes requires care to avoid covert channels. One simple way to avoid such channels is to use fewer error codes (or drop error codes altogether), but doing so makes debugging applications difficult.

NiStar addresses this issue by performing flow checks as early as possible. For example, many system calls need to check whether the current thread has permission to access specified data. After such a flow check succeeds, the system call has more liberty to validate parameters and return more specific error codes without violating noninterference.

LIMIT RESOURCE USAGE WITH QUOTAS. Shared resources can lead to covert channels due to resource exhaustion. Systems may impose a quota on shared resources for each domain to avoid such channels. There are several quota schemes. One simple scheme is to statically assign predetermined quotas to domains; for instance, allowing processes to allocate only a predetermined number of identifiers for child processes [31]. However, this scheme limits the functionality of the system if the quota is too low, and wastes resources if the quota is set too high.

A more flexible and explicit quota scheme is to organize resources into a hierarchy of *containers* [16, 145, 169], where each container has a quota for resources such as memory and CPU time. A thread can allocate objects from a container, including creating subcontainers, if the container has sufficient quota and the policy allows the thread to access the container. A thread can also transfer quotas between two containers if the policy allows the thread to access both containers. NiStar uses containers to manage resources.

PARTITION NAMES AMONG DOMAINS. Resource names in a shared namespace, such as thread identifiers and page numbers, can lead to covert channels. A per-domain naming scheme partitions names among domains to eliminate such channels. A classical example is using ⟨process identifier, virtual page number⟩ pairs to refer to memory pages, effectively partitioning page numbers among processes. As another example, a system with container-based resource management may use ⟨container identifier, resource identifier⟩ pairs to refer to resources [169]; a thread may access the resource only if the policy permits it to access the container. Both NiStar and NiKOS employ per-domain naming schemes.

ENCRYPT NAMES FROM A LARGE SPACE. Using encrypted names is an alternative way to address covert channels due to resource names. Many DIFC systems allocate sequential identifiers for resources, but return *encrypted* values to make them unpredictable [37, 84, 169]. This design technically violates noninterference, but since the identifier space is sufficiently large (e.g., 64 bits), the amount of information that can be leaked through this channel is negligible in practice. However, verifying noninterference for this design would require probabilistic reasoning [83] and complicate the semantics of noninterference [39, §6.4]. We therefore do not use encrypted names for the systems verified using Nickel.

EXPOSE OR ENCLOSE NONDETERMINISM. As mentioned in Section 5.2.5, Nickel does not allow nondeterministic behavior in the interface specification (for instance, a system call that allocates an unspecified physical page), since doing so would complicate refinement for noninterference.

There are several options for revising the semantics of such system calls to eliminate nondeterminism. The first option is to make the (nondeterministic) decision explicit as a system call parameter, for example, asking user space to decide which page to allocate, similarly to exokernels [41, 75]. The second option is to ask developers to explicitly describe the behavior (e.g., the allocation algorithm) as part of the interface specification. This makes the interface specification less abstract but simplifies the verification of noninterference under refinement; NiStar uses this option for memory management. The third option is to enclose the source of nondeterminism below the interface [56], for example, using virtual addresses to refer to memory pages and removing the use of physical pages from the interface. NiKOS uses this option.

REDUCE FLOWS TO THE SCHEDULER. An OS scheduler is generally associated with a powerful domain, such as in Figure 19. The scheduler decides and updates which process to run, and other domains usually need to access this information (e.g., to look up the process currently running), creating inherent flows from the scheduler to other domains. Many scheduling approaches access information about processes to make scheduling decisions, creating flows from other domains to the scheduler. The combination of these flows makes the scheduler a powerful domain that two processes might exploit to communicate.

One way to control this risk is to enforce a stricter policy that prohibits flows *to* the scheduler. This policy restricts the power of the scheduler, since it can no longer query state that belongs to other domains. One simple design that satisfies this policy is to use a static, predetermined schedule [7, 113] that does not need to query the system state for scheduling decisions. NiStar instead satisfies this policy with a more flexible design: like exokernels [41, 75], it allows applications to allocate time slices to implement dynamic scheduling policies. Unlike exokernels, NiStar performs flow checks at run time to prevent these allocations introducing covert channels (see Section 5.5.2).

5.5 DIFC IN NISTAR

NiStar is a new OS kernel that supports decentralized information flow control (DIFC). NiStar's design is inspired by HiStar [169]: the kernel tracks information flow using labels and enforces DIFC through seven object types, and a user-space library implements POSIX abstractions on top of these kernel object types. Unlike HiStar, however, we have formalized NiStar's information flow policy and verified that both its interface specification and implementation satisfy noninterference for this policy. This section describes how we designed the NiStar interface to eliminate covert channels and used Nickel to achieve automated verification.

5.5.1 *Labels*

Like other DIFC systems [51, 84, 135], NiStar uses tags and labels to track information flow across the system. It follows a scheme used in DStar [168] and a revised version of HiStar [170]. A tag is an opaque integer, which has no inherent meaning. For instance, Alice uses tags t_S and t_I to represent the secrecy and integrity of her data, respectively. A label is a set of tags. Every object in the system is associated with a triple of $\langle \text{secrecy}, \text{integrity}, \text{ownership} \rangle$ labels, which we designate as the domain of the object. For instance, Alice labels her files with $\langle \{t_S\}, \{t_I\}, \varnothing \rangle$.

We use Figure 24 as an example to illustrate how Alice can constrain untrusted applications using labels. Suppose Alice launches a spellchecker to scan her files; the spellchecker consults a shared dictionary and prints the results (misspelled words) to her terminal. An updater periodically queries a server through the `netd` daemon and keeps the dictionary up to date. Alice trusts her `ttyd` daemon to declassify data only to her terminal. She trusts neither the spellchecker nor the updater, which may each be buggy, compromised, or malicious. Alice hopes to achieve the following security goals: (1) neither the spellchecker nor the updater can modify her files; and (2) her spellchecked files can not be leaked to the network.

Classical information flow control expresses policies using only secrecy and integrity labels (i.e., ignoring ownership). Given two objects with domains $L_1 = \langle S_1, I_1, O_1 \rangle$ and $L_2 = \langle S_2, I_2, O_2 \rangle$, respectively, it is safe in the classical model for information to flow from L_1 to L_2 if (1) the secrecy of S_1 is subsumed by that of S_2 and (2) the integrity of I_1 subsumes that of I_2: $S_1 \subseteq S_2 \wedge I_2 \subseteq I_1$. In other words, a flow is safe if it neither discloses secrets nor compromises the integrity of any object. For example, given the label assignment in Figure 24 and a system enforcing such flow checks, Alice can conclude that her files will not be modified by the spellchecker or the updater: her files have

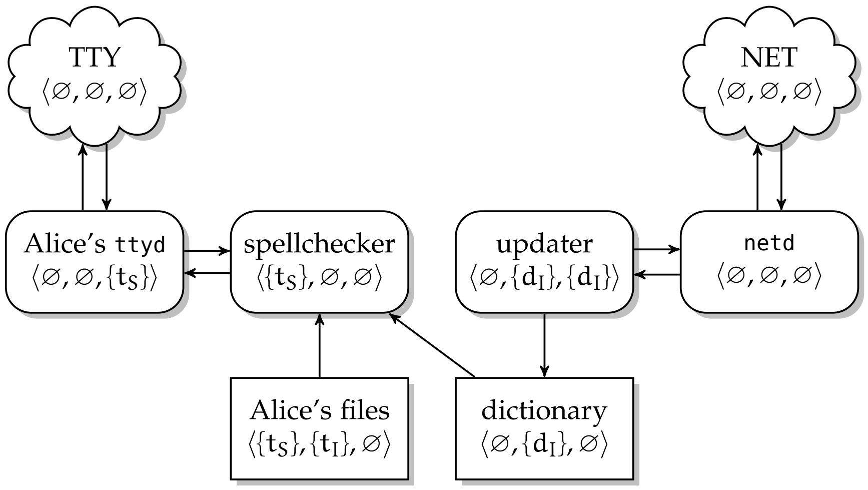


Figure 24: Information flow of a spellchecker and updater. Cloud boxes represent terminal (TTY) and network (NET); rounded boxes represent threads; and rectangular boxes represent data. Each object is associated with a triple of ⟨secrecy, integrity, ownership⟩ labels; arrows denote the flows of information allowed by these labels.

t_I in their integrity labels, but the spellchecker and updater do not, ruling out flows from them to her files.

The classical model is often too restrictive for practical systems. For instance, a password checker needs to declassify whether login succeeds to untrusted users; as another example, to output misspelled words in Figure 24, the spellchecker (with t_S in secrecy) needs to communicate with to Alice's trusted `ttyd` (without t_S). Like other DIFC systems, NiStar supports such intentional downgrading without a centralized authority. It uses the ownership label to relax label checking for trusted threads, giving them the privilege to temporarily remove tags from secrecy labels (declassification) or add tags to integrity labels (endorsement), as follows:

Definition 5 (Safe Flow)**.** Information can flow from $L_1 = \langle S_1, I_1, O_1 \rangle$ to $L_2 = \langle S_2, I_2, O_2 \rangle$, denoted as $L_1 \rightsquigarrow L_2$, if and only if $(S_1 - O_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1 \cup O_1)$.

This can-flow-to relation is central to NiStar's information flow policy. $L_1 \rightsquigarrow L_2$ means that L_1 and L_2 can combine their ownership to allow the maximum flow from L_1 to L_2; that is, L_1 lowers its secrecy to $S_1 - O_1$ and raises its integrity to $I_1 \cup O_1$, while L_2 raises its secrecy to $S_2 \cup O_2$ and lowers its integrity to $I_2 - O_2$.

Referring to Figure 24, as information can flow from the spellchecker to Alice's `ttyd` given their label assignments, $\langle \{t_S\}, \varnothing, \varnothing \rangle \rightsquigarrow \langle \varnothing, \varnothing, \{t_S\} \rangle$, Alice's `ttyd` is able to print out misspelled words. In addition, Alice can conclude that her files will not be leaked to the network: the spellchecker cannot directly leak information to the network given its label assignment. The spellchecker can, however, indirectly write to

Alice's terminal only through her `ttyd`, which she trusts to declassify data only to the terminal; no other threads in the system are trusted. This example shows how labels can minimize the amount of application code that must be trusted.

5.5.2 *Kernel objects*

NiStar provides seven object types:

- *labels* represent domains of objects;
- *containers* are basic units for managing resources;
- *threads* are basic execution units;
- *gates* provide protected control transfer;
- *page-table pages* organize virtual memory;
- *user pages* represent application data; and
- *quanta* represent time slices for scheduling.

Each object, other than labels, is associated with a domain of $\langle \text{secrecy}, \text{integrity}, \text{ownership} \rangle$ labels; only threads and gates can have non-empty ownership labels. The kernel interface consists of a total of 46 operations for manipulating these objects. Each operation performs flow checks among objects using their labels. NiStar's design goal is to ensure that the interface specification satisfies noninterference for the policy given by 5.

NiStar largely follows HiStar's object types [169], with the following exceptions: it provides a new object type, quantum, for scheduling; and to make the interface finite and therefore amenable to automated verification, it uses fixed-sized page-table pages and user pages similar to Hyperkernel (Chapter 4) and seL4 [78]. Interested readers can refer to Zeldovich et al. [170] for details of object types and label checks; below, we highlight three key differences in NiStar that close covert channels.

Given $L_1 = \langle S_1, I_1, O_1 \rangle$ and $L_2 = \langle S_2, I_2, O_2 \rangle$, we introduce the following notations for flow checks:

- $L_1 \sqsubseteq_R L_2$ means that L_1 *can be read by* L_2: $(S_1 \subseteq S_2 \cup O_2) \land (I_2 - O_2 \subseteq I_1)$.
- $L_1 \sqsubseteq_W L_2$ means that L_1 *can write to* L_2: $(S_1 - O_1 \subseteq S_2) \land (I_2 \subseteq I_1 \cup O_1)$.

As a shorthand, we write $L_2 \sqsubseteq_R L_1 \sqsubseteq_W L_2$ to mean that L_1 *can modify* L_2: $(L_2 \sqsubseteq_R L_1) \land (L_1 \sqsubseteq_W L_2)$. It is generally difficult for L_1 to modify L_2 without receiving any information in return (e.g., error code), and so this definition includes L_1 being able to read L_2. By definition, $L_1 \sqsubseteq_W L_3$ and $L_3 \sqsubseteq_R L_2$ together imply $L_1 \rightsquigarrow L_2$ for any L_1, L_2, and L_3; we will use this fact below to analyze covert channels. We denote $\mathcal{L}_x$ as the domain of object x.

MAINTAIN ACCURATE QUOTAS IN CONTAINERS. Like HiStar, NiStar manages all system resources in a hierarchy of containers, starting from a root container created during kernel initialization. Each container maintains a set of quotas, indicating the amount of memory pages and time quanta it owns. A thread T may allocate an object O from a container C only if it can modify the container (i.e., $\mathcal{L}_C \sqsubseteq_R \mathcal{L}_T \sqsubseteq_W \mathcal{L}_C$), the new object does not exceed the authority of the thread (i.e., $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$), and the container has sufficient quota for the object.

NiStar maintains accurate quotas in containers, which differs from HiStar in two ways. First, NiStar sets the memory quota of the root container to be number of available physical pages upon booting, rather than infinity [169, §3.3], avoiding a potential covert channel due to resource exhaustion. Second, NiStar does not allow an object to be linked by multiple containers, which would require the kernel to conservatively charge each container as in HiStar. Instead, each object is uniquely owned by one container. This design leads to a simpler invariant: for each resource type, the sum of the quotas of each object in a container equals the total quota of the container.

ENFORCE CAN-WRITE-TO-OBJECT ON DEALLOCATION. In HiStar, to deallocate an object O from a container C, a thread T must be able to write to the container, but not necessarily to the object itself. This relaxed check supports reclaiming *zombie* objects to which no one else can write (e.g., those with a unique integrity tag) [167]. However, it leads to a covert channel. Consider a thread T′ whose domain permits it to read object O (i.e., $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$) but prohibits it from receiving information from thread T (i.e., $\mathcal{L}_T \not\rightsquigarrow \mathcal{L}_{T'}$). To bypass DIFC, thread T encodes a one-bit secret by either deallocating object O from container C or not. T′ learns the secret by observing whether object O still exists [169, §3.2], violating noninterference since the label assignment prohibits information flow from T to T′.

NiStar enforces a stricter flow check on deallocation by requiring that thread T can write to object O (i.e., $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$). With this stricter check, this covert channel is closed: if thread T′ can read object O (i.e., $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$), the new check implies that thread T′ is permitted to receive information from thread T, since $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_O$ and $\mathcal{L}_O \sqsubseteq_R \mathcal{L}_{T'}$ together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$.

NiStar considers reclaiming zombie objects an administrative decision and leaves it to user space. Some systems may consider it legitimate for a user to create objects that no one else can reclaim; since NiStar enforces accurate quotas, adversarial users cannot create "runaway" zombie objects that exceed their quotas. On the other hand, a system wishing to reclaim zombie objects can emulate the HiStar behavior by setting up a trusted garbage collector with a powerful domain during booting, without baking this requirement into flow checks in the kernel.

REMOVE FLOWS TO THE SCHEDULER USING QUANTA. As noted in Section 5.4, two processes can exploit the scheduler to communicate in violation of information flow policy. To close this channel, NiStar borrows the design of the exokernel scheduler [41] and extends it with label checking. NiStar associates the scheduler with domain $\langle\varnothing, \mathbb{U}, \varnothing\rangle$, where $\mathbb{U}$ denotes the universal label of all tags. This domain allows the scheduler to switch to any thread (its universal integrity allows it to influence any thread it runs) while restricting it from leaking information (its empty secrecy and ownership prevent it receiving secrets). The resulting scheduler allows applications to implement more flexible scheduling schemes compared to static scheduling.

NiStar introduces *time quanta* to allow the scheduler to make decisions while respecting this label assignment. The system is configured with a fixed number of quanta, each associated with a thread identifier for scheduling. Like other resources, all quanta are initially owned by the root container; a thread can move quanta between two containers only if it can modify both containers. To schedule thread T' at quantum Q, thread T writes the identifier of T' to Q. Thread T can perform this write only if it can write to quantum Q (i.e., $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_Q$).

To schedule using time quanta, assume that the system delivers an infinite stream of timer interrupts. Upon the arrival of a timer interrupt, the scheduler cycles through all the quanta in a round-robin fashion and retrieves the thread identifier T' associated with the next quantum Q. If quantum Q can be read by thread T' (i.e., $\mathcal{L}_Q \sqsubseteq_R \mathcal{L}_{T'}$), the scheduler switches to T'; otherwise, it idles.

To see why these flow checks suffice to close the channel, suppose T is able to schedule T' to execute at quantum Q. The checks ensure $\mathcal{L}_T \sqsubseteq_W \mathcal{L}_Q$ and $\mathcal{L}_Q \sqsubseteq_R \mathcal{L}_{T'}$, which together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$; in other words, the label assignment permits T to communicate with T'.

This design closes covert channels arising from logical time. As mentioned in Section 5.2.5, physical timing is beyond the scope of this thesis, for which NiStar provides no guarantees of noninterference.

5.5.3 *Implementation*

To demonstrate that NiStar's interface is practical, we have built a prototype implementation for x86-64 processors, and have applied Nickel to verify that both the interface specification and the implementation satisfy noninterference for the policy given by 5.

To simplify verification, NiStar borrows ideas from previous verified OS kernels. First, like Hyperkernel (Chapter 4), NiStar uses separate page tables for the kernel and user space. It uses an identity mapping for the kernel address space, sidestepping the complication of reasoning about virtual memory for kernel code [81]. Second, like seL4 [78], NiStar enables timer interrupts only in user space and disables them in the kernel. This restriction ensures that the execution of system calls and exception handling is atomic, avoiding reasoning about interleaved executions. Third, NiStar disables all other interrupts and requires device drivers to use polling, a common practice in high-assurance systems [7, 113].

For user space, we have ported the *musl* C standard library [1] to NiStar, running on top of an emulation layer for Linux system calls. A library implements the abstraction of Unix-like processes on top of NiStar's kernel object types, similar to HiStar's emulation layer [169]. The file-system service is implemented as a thin wrapper over containers and user pages, and the network service is provided by lwIP [35]. Although our current user space implementation is incomplete, it is able to run programs such as a set of POSIX utilities from Toybox, a web server, and the TinyEMU emulator to boot Linux.

5.6 VERIFYING ISOLATION

Nickel generalizes to information flow control systems beyond DIFC. This section describes applying Nickel to two such systems: NiKOS and ARINC 653.

PROCESS ISOLATION. NiKOS is a small OS that enforces an isolation policy among processes (Figure 19). The interface of NiKOS mirrors that of a version of mCertiKOS as described by Costanzo, Shao, and Gu [31]. It consists of seven operations, including spawning a process, querying process status, printing to console, yielding, and handling a page fault. Like mCertiKOS, NiKOS imposes a memory quota on each process and statically partitions identifiers among processes, avoiding covert channels due to resource names and exhaustion (Section 5.4). We implemented a prototype of NiKOS for x86-64 processors and ported user-space applications from mCertiKOS. We used Nickel to verify that both the interface and implementation satisfy noninterference for the isolation policy. This effort took one author a total of two weeks.

We made one change to the design in order to verify noninterference. In mCertiKOS, the `spawn` system call creates a new process and loads an executable file; the specification of `spawn` models file loading as a no-op, whereas the implementation allocates pages and consumes memory quota [54]. In NiKOS, to match the memory quota in the specification with that in the implementation, `spawn` creates an empty address space and the page-fault handler lazily loads each page of the executable file instead.

PARTITION ISOLATION. ARINC 653 [7] is an industrial standard for safety-critical avionics operating systems. It models the system as a set of *partitions* and defines an inter-partition communication interface comprising 14 operations. Figure 25 depicts its isolation policy among partitions: information can flow to a partition only from the

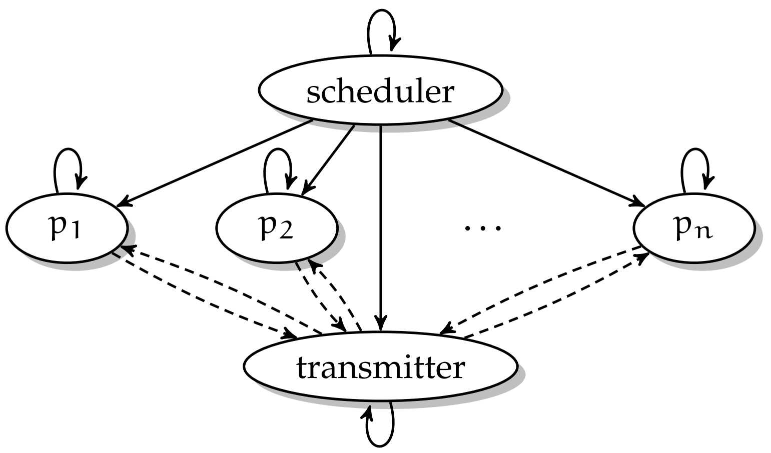


Figure 25: The isolation policy of ARINC 653: information can flow between the transmitter and each partition p_i for $i \in [1, n]$ as per a boot-time configuration (dashed arrows); it cannot flow between any two partitions, or from any partition or the transmitter to the scheduler.

transmitter, the scheduler, and itself. The transmitter forwards messages among partitions as configured at boot time; each dashed arrow represents a flow that can be independently enabled in the configuration. The scheduler uses a pre-configured fixed schedule, and so does not require flows from other domains to the scheduler (Section 5.4).

Using Nickel, we formalized the specification of the communication interface based on the pseudocode provided by the ARINC 653 standard. Applying Nickel to verify noninterference for the partition isolation policy reproduced all three known covert channels first discovered by Zhao et al. [173], which were caused by missing partition permission checks, allocating identifiers in a shared namespace, and returning error codes that leak information; verification succeeded once we fixed these channels. This effort took one author a total of one week.

5.7 EXPERIENCE

This section reports our experience with using Nickel and reflects lesson learned during development. Experiments ran on an Intel Core i7-7700K CPU at 4.5 GHz.

COVERT CHANNEL DISCUSSION. To test the effectiveness of Nickel for detecting covert channels, we injected each of the examples in Section 5.1 into the NiStar interface specification. In each case, Nickel was able to find a counterexample pointing to the issue. As a concrete example, we switched NiStar's scheduler to a round-robin one. When verifying this round-robin scheduler, Nickel failed and produced a counterexample (Section 5.3).

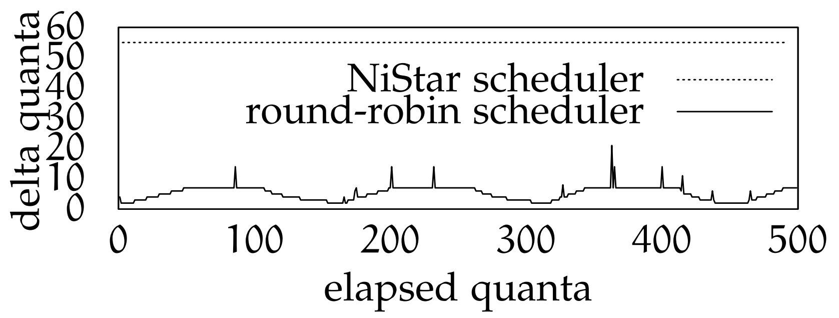


Figure 26: A round-robin scheduler leaks background thread behavior through patterns in logical time; no such pattern is observed in NiStar.

Figure 26 shows empirical evidence of a covert channel by comparing the NiStar scheduler with the round-robin one. In this experiment, one process sampled the current (logical) time, while a background process repeatedly forked and then killed 30 child processes. The measuring process recorded the duration between scheduling points in terms of number of quanta. With the round-robin scheduler, the gaps observed by the measuring process vary as the background task forks and kills its children, creating patterns that indicate the covert channel. With the NiStar scheduler, which is verified using Nickel, the gaps between scheduling points remain constant regardless of the behavior of the background process. This result suggests that the Nickel is effective in identifying and proving the absence of covert channels.

DEVELOPMENT EFFORT USING NICKEL. Figure 27 shows the sizes of the three systems we verified using Nickel: NiStar, NiKOS, and ARINC 653. The lines of code for the interface implementations of both NiStar and NiKOS do not include common kernel infrastructure (C library functions and x86 initialization), and those of the user space implementations do not include third-party libraries (e.g., musl and lwIP). The implementation of the Nickel framework is split between the formalization of the metatheory (1,215 lines of Coq) and the verifier for the unwinding and refinement conditions (3,564 lines of C++ and Python).

The information flow policies for the three systems are concise compared to the rest of the specification and implementation, indicating the simplicity of creating policies ranging from DIFC to isolation using Nickel (Section 5.3).

In our experience, the most time-consuming part of the verification process was coming up with an appropriate observational equivalence relation—it was non-trivial to determine which part of the system state was observable by each domain, and the complexity increased as the size of the system state and the number of interface operations grew. We found the counterexamples produced by

component	NiStar	NiKOS	ARINC 653
specification:			
information flow policy	26	14	33
interface specification	714	82	240
proof input:			
interface invariant	398	63	66
observational equivalence	127	56	80
implementation invariant	52	7	–
data refinement	139	30	–
implementation:			
interface implementation	3,155	343	–
user space implementation	9,348	389	–
common kernel infrastructure	4,829 (shared by NiStar/NiKOS)		

Figure 27: Lines of code for the three systems verified using Nickel.

Nickel particularly useful for debugging and fixing observational equivalence. The specification and verification of NiStar, NiKOS, and ARINC 653 took one author six weeks, two weeks, and one week, respectively; as a comparison, implementing NiStar took several researchers roughly six months. This comparison shows that the proof effort required when using Nickel is low, thanks to its support for automated verification and counterexample generation.

Using Z3 4.6.0, verifying NiStar, NiKOS, and ARINC 653 on four cores took 72 minutes, 7 seconds, and 8 seconds, respectively.

LESSONS LEARNED. Our development of Nickel was guided by two motives. First, in our previous work on Hyperkernel described in Chapter 4, we proved memory isolation among processes, but this did not preclude covert channels through system calls; Nickel extends push-button verification to support proving stronger guarantees about noninterference. Second, we aimed to develop a general framework that can help analyze and design interfaces not only for isolation, but also for mechanisms as flexible as DIFC.

While designing Nickel, we spent a total of two months iterating through several formulations of noninterference before settling on the one described in Section 5.2. Among these alternatives were classical transitive noninterference [58] and intransitive noninterference [138], as well as variants such as nonleakage [112, 119]. As discussed in Section 5.2.5, Nickel's formulation has the advantage of supporting both a spectrum of policies and automated verification.

As Figure 23 shows, Nickel combines both automated and interactive theorem provers: Z3 automates proofs for individual systems, while the proofs in Coq improve confidence in Nickel's metatheory. Similar approaches have been used for the verification of compiler optimizations [147], static bug checkers [155], and Amazon's s2n TLS library [26]. We believe that this combination is an effective approach to developing verified systems.

5.8 RELATED WORK

VERIFYING NONINTERFERENCE IN SYSTEMS. Noninterference is a desirable security definition for operating systems looking to guarantee information flow properties [137]. For example, the seL4 microkernel [78] is proven to satisfy a variant of noninterference for a given access control policy [112, 113]; a version of mCertiKOS [55] includes a proof of process isolation [31]; Ironclad [61] proves end-to-end guarantees for applications using a form of input and output noninterference; and Komodo [45] proves noninterference for isolated execution of software-based enclaves. Noting the difficulty of extending noninterference proofs to concurrent systems, Covern [111] provides a logic for the shared memory setting. Noninterference also has applications in secure hardware [46, 47], programming languages [96, 148], as well as browsers and servers [70, 131]. Nickel takes inspiration from these efforts, focusing on formalizations and interface designs that are amenable to automated verification of noninterference.

DIFC OPERATING SYSTEMS. Information flow control was originally envisioned as a mechanism to enforce multi-level security in military systems [13, 15]. *Decentralized* information flow control (DIFC) additionally allows applications to declare new classifications [114, 115]. The design of NiStar was influenced by prior DIFC operating systems [24, 37, 84, 135, 169], particularly HiStar and Flume.

HiStar [169, 170] enforces DIFC with a small number of types of kernel objects. All label changes in HiStar are explicit, closing the covert channel in Asbestos due to implicit label changes [37]. NiStar's design draws from HiStar, using a similar set of kernel object types, but adapted to close remaining covert channels and enable automated verification.

Flume [84] is a DIFC system built on top of the Linux kernel. Building on top of an existing kernel makes porting easier, but expands Flume's TCB. Flume's design has a pen-and-paper proof [83] of noninterference for a single label assignment, modeled using Communicating Sequential Processes [66]; a more general formalization of Flume is given by Eggert [38]. NiStar takes this effort a step further, with the first noninterference proof of both the interface and implementation of a DIFC OS kernel.

REASONING ABOUT INFORMATION FLOWS FOR APPLICATIONS. Assigning DIFC labels for applications is a non-trivial task. To help application developers, Asbestos offers a domain-specific language [36] for generating label assignments from high-level specifications. The SWIM tool [59] generates label assignments from lists of prohibited and allowed flows, and has been further extended using synthesis techniques [60]. These tools can benefit from a precise specification of the DIFC framework they use to implement policies for, such as the one provided by NiStar.

5.9 CONCLUSION

Nickel is a framework for designing and verifying information flow control systems through automated verification techniques. It focuses on helping developers eliminate covert channels from interface designs and provides a new formulation of noninterference to uncover covert channels or prove their absence using an SMT solver. We have applied Nickel to develop three systems, including NiStar, the first formally verified DIFC OS kernel. Our experience shows that the proof burden of using Nickel is low. We believe that Nickel offers a promising approach to the design and implementation of secure systems. All of Nickel's source code is publicly available at `https://unsat.cs.washington.edu/projects/nickel/`.

6 RATATOSKR: FUTURE WORK AND PRELIMINARY RESULTS ON PUSH-BUTTON VERIFICATION OF DISTRIBUTED SYSTEMS

This chapter describes preliminary and future work on push-button verification of distributed systems.

Designing and building systems for failure is crucial to ensure reliable operation and availability when the inevitable happens: hardware fails or software bugs take down part of the system. A common approach to providing robustness against failures is through state machine replication [88, 143]. The idea is to implement the systems as a deterministic state machine and run it on multiple nodes to provide the desired level of redundancy. As long as a majority of the nodes are live, the system remains available. An essential requirement for such systems is consensus: for all the nodes to compute the same output, they must agree on the sequence of inputs. However, distributed systems can exhibit a significant degree of nondeterminism due to network semantics and failures. This nondeterminism makes the protocol hard to reason about and implement correctly. Protocol and implementation bugs in distributed systems are notoriously hard to find and reproduce and a common cause of data loss, inconsistencies, or service unavailability [163].

This chapter presents Ratatoskr: a protocol specification and proof of correctness for the Disk Paxos algorithm [48]. Disk Paxos is a variant of Paxos [86], a widely used algorithm for achieving consensus. We have verified that the Ratatoskr protocol specification correctly achieves consensus, using the Z3 automated theorem prover. The Ratatoskr proof uncovered a bug in the algorithm as described by Gafni and Lamport [48].

We leave refining the protocol to a working implementation as future work.

6.1 OVERVIEW AND BACKGROUND

This section provides an overview of the Ratatoskr verification, a quick overview of the Disk Paxos algorithm, and how Ratatoskr models it.

DISK PAXOS Disk Paxos [48] is a variant of the Paxos [86] algorithm achieving consensus on multiple nodes for fault tolerance. Disk Paxos uses two types of nodes: processor nodes and disk nodes. Processor nodes process requests and execute the state machine but store no persistent state. In contrast, disk nodes perform no computation, only providing persistent for the processor nodes.

The Disk Paxos algorithm tolerates non-Byzantine failures. A processor can pause for an arbitrarily long time or crash, losing all its volatile state. Disks may become inaccessible to some or all processors, but it may not lose writes it has acknowledged. The algorithm remains live, processing requests as long as a majority of disks and at least one processor node is up.

Similar to Paxos, Disk Paxos commits a single value at a time. To commit a sequence of values (e.g., the sequence of inputs to a replicated state machine) multiple separate *instances* of the algorithm are used. Below we describe a single instance of Disk Paxos. Processor nodes store the triple $\langle \mathrm{mbal}, \mathrm{bal}, \mathrm{inp} \rangle$ and each disk stores $\langle \mathrm{mbal}, \mathrm{bal}, \mathrm{inp} \rangle$ for *every* processor node. For a processor p, mbal represents the current ballot number, bal is the largest ballot number it has entered phase 2, and inp is the value p tried to commit with bal. The Disk Paxos algorithm transitions through 4 phases. Each phase is briefly described below.

PHASE 0. A processor reads its state from a quorum of disks blocks, recovering any state it may have lost during a failure.

PHASE 1. A processor proposes to commit an input with a new ballot number by writing to a quorum of disks and reading the corresponding disk block for every other processor. If a processor reads a block with a larger ballot (mbal) number, it aborts and retries with a larger number.

PHASE 2. A processor tries to commit an input with the ballot number from the previous phase by writing the input associated with the highest ballot number seen in phase 1, or the user supplied value if no other input was seen. After writing to a quorum of disks, and reading every other processor's blocks from a quorum of disks without seeing a larger ballot number, the processor can transition to phase 3.

PHASE 3. The processor has either committed a new value or learned of previously committed value.

VERIFICATION Our goal is to create a specification that captures the Disk Paxos protocol and to prove its correctness: that it achieves consensus. We leave as future work to implement the protocol and prove that it refines the specification.

The verification of Ratatoskr proves the following theorem:

Theorem 12 (Ratatoskr consistency)**.** If a processor node in Ratatoskr commits a value v for some instance i, then every value committed in i by any processor equals v.

We write the specification of the Disk Paxos protocol in a subset of the Python programming language, which can be compiled to SMT expressions and sent to the Z3 [109] solver. Next, we describe our execution model and formalize our definition of consistency.

6.2 PROVING CONSISTENCY

The Ratatoskr verifier models the system $\mathcal{M}$ as a state machine. State transitions occur when a processor application invokes any defined operation in the Ratatoskr interface.

Formally we define

$$\mathcal{M} = \langle \mathcal{P}, \mathcal{D}, \mathcal{S}_{\mathcal{P}}, \mathcal{S}_{\mathcal{D}}, \mathcal{A}, \mathcal{V}, \mathcal{I}, \texttt{init}, \texttt{step}, \texttt{output} \rangle$$

where $\mathcal{P}$ is the set of processor nodes; $\mathcal{D}$ is a set of disk nodes; $\mathcal{S}_{\mathcal{P}}$ is the set of processor states; $\mathcal{S}_{\mathcal{D}}$ is the set of disk states; $\mathcal{A}$ is the set actions; $\mathcal{V}$ is a set of input values; $\mathcal{I}$ is a set of instances; $\texttt{init} : \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{D}}$ is the initial processor and disk states; $\texttt{step} : \mathcal{P} \times (\mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{D}}) \times \mathcal{A} \rightarrow \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{D}}$ is the transition function, taking a processor, processor and disk states, an instance, and an action, and providing a new processor and disk state; and finally $\texttt{output} : \mathcal{P} \times \mathcal{S}_{\mathcal{P}} \times \mathcal{I} \rightarrow \mathcal{V} \cup \{\bot\}$ outputs the value learned by processor in a particular instance, or a special bottom value.

A predicate is invariant over a state machine if it always holds for any execution of the system starting in the initial state. Formally, we define it as follows

Definition 6 (Invariant)**.** We say that a predicate I is invariant for a system $\langle \mathcal{P}, \mathcal{D}, \mathcal{S}_{\mathcal{P}}, \mathcal{S}_{\mathcal{D}}, \mathcal{A}, \mathcal{V}, \mathcal{I}, \texttt{init}, \texttt{step}, \texttt{output} \rangle$ if $I(\texttt{init})$ and

$$\forall p, s, d, a.\ I((s, d)) \Rightarrow I(\texttt{step}(p, (s, d), a))$$

To check consistency of Ratatoskr, the verifier ensures that once a value has been committed for some instance, it must stay committed and a different value may never be committed for that instance. Given a relation $\texttt{chosen} : \mathcal{S}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{D}} \times \mathcal{I} \times \mathcal{V}$, relating a system state and instance number to a value v; we say that v is chosen, and that the system respects the relation if the following condition holds:

- $\forall s, d, i, v1, v2.$
  $\texttt{chosen}((s, d), i, v1) \wedge \texttt{chosen}((s, d), i, v2) \Rightarrow v1 = v2$
- $\forall p, s, d, i, a, v.$
  $\texttt{chosen}((s, d), i, v) \Rightarrow \texttt{chosen}(\texttt{step}(p, (s, d), a), i, v)$

That is, no two different values can be chosen at the same time, and if a value is chosen in some state, then it will be always be chosen (no state transition changes the chosen value).

We now define consistency as follows

Definition 7 (Consistency)**.** We say that a system

$$\mathcal{M} = \langle \mathcal{P}, \mathcal{D}, \mathcal{S}_{\mathcal{P}}, \mathcal{S}_{\mathcal{D}}, \mathcal{A}, \mathcal{V}, \mathcal{I}, \texttt{init}, \texttt{step}, \texttt{output} \rangle$$

is consistent if there exists an invariant I for $\mathcal{M}$, a relation `chosen` for which $\mathcal{M}$ respects and

$$\forall p, s, d, i.\ o = \bot \lor \mathrm{chosen}((s, d), i, o)$$
$$\quad \text{where } o = \mathrm{output}(p, s, i)$$

6.3 VERIFYING RATATOSKR

The Ratatoskr verifier proves Theorem 12 by requiring the developer to supply a `chosen` relation and specification invariants that satisfy Definition 7, ensuring that if *any* processor outputs value o in some state, then o must also be chosen. Consequently, given that the system respects `chosen`, no two processors can output different values at the same time, nor can the output value ever change.

Next we describe how Ratatoskr models the Disk Paxos algorithm for automated verification, the state, the set of protocol operations defined in the Ratatoskr interface, and the invariants proved.

THE RATATOSKR STATE Ratatoskr models the Disk Paxos state using uninterpreted functions and six uninterpreted sorts: Ballot, Instance, Value, Proc (processor id), Disk (disk id) and Quorum. Designating a special $\bot$ of type Value to mean no value. Phase is a bitvector between 0 and 3. The full Ratatoskr state is shown in Figure 28.

Ratatoskr makes only two assumptions about those types. First, a correctness requirement for Disk Paxos is that each node produces unique and increasing ballot numbers. This is commonly achieved by labeling each processor with a unique value between 1 and n, where n is the number of processes. Each processor then increases their ballot number by n. Ratatoskr instead axiomatizes a total order and a set partition on Ballot (a cell for each processor), allowing any implementation. Second, Ratatoskr axiomatizes a relation member : Disk $\times$ Quorum such that for every pair of quorum, there exists a disk that is a member of both.

QUANTIFIER ELIMINATION Ratatoskr explicitly keeps track of the read/write quorum for each process as they transition through the phases of the algorithm, requiring processor nodes to explicitly pass in the read/write quorum to transition to the next phase. This reduces quantifier alternations in our queries (i.e., it removes the need for existential quantification of quorums by naming them explicitly instead), making verification significantly faster.

Processor state:

$$\begin{aligned}
\text{proc\_mbal} &: \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Ballot} \\
\text{proc\_bal} &: \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Ballot} \\
\text{proc\_input} &: \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Value} \\
\text{proc\_output} &: \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Value} \\
\text{proc\_phase} &: \text{Proc} \rightarrow \text{Phase}
\end{aligned}$$

Disk state:

$$\begin{aligned}
\text{disk\_mbal} &: \text{Disk} \rightarrow \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Ballot} \\
\text{disk\_bal} &: \text{Disk} \rightarrow \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Ballot} \\
\text{disk\_input} &: \text{Disk} \rightarrow \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Value}
\end{aligned}$$

Global ghost state:

$$\begin{aligned}
\text{quorum} &: \text{Proc} \rightarrow \text{Instance} \rightarrow \text{Phase} \rightarrow \text{Quorum} \\
\text{chosen} &: \text{Instance} \rightarrow \text{Value}
\end{aligned}$$

Figure 28: The Ratatoskr state, composed of three records. One for the processors, one for the disks and a global ghost state.

```
get_phase(instance)                     phase1_request_read(instance, disk, proc)
get_self_id()                           phase1_request_write(instance, disk)
get_num_procs()                         phase1_end(instance, quorum, value)
get_num_disks()                         phase2_start_ballot(instance)
phase0_request_read(instance, disk)     phase2_request_read(instance, disk, proc)
phase0_end(instance, quorum)            phase2_request_write(instance, disk)
phase1_start_ballot(instance)           phase2_end(instance, quorum)
                                        phase3_get_output(instance)
```

Figure 29: Complete interface for Ratatoskr.

OPERATIONS Ratatoskr exposes a total of 15 operations. These operations are invoked by processor nodes and drive the consensus protocol. Most of those operations, 10 in total, transition an instance through each of the four phases of the Disk Paxos algorithm. Each phase has a set of operations enabled to move the protocol forward. Disabled actions are treated as a NO-OP, leaving the state unchanged. The full set of operations is shown in Figure 29.

INVARIANTS The Ratatoskr verifier checks a total of 39 invariants for each of the operations in Ratatoskr. Z3 is unable to directly prove the invariants described by Gafni and Lamport [48], requiring substantial changes. The two main changes are summarized next. First, Ratatoskr uses uninterpreted functions as a core primitive, requiring additional invariants compared to sets. Second, a deep nesting of quantifier alternations caused timeouts. So to scale verification, Ratatoskr explicitly names quorums, eliminating the need for nested existential quantifiers, adding additional invariants relating read and write sets to the quorum function in the global state.

Together, the Ratatoskr invariants imply Definition 7, concluding the proof of Theorem 12.

6.4 EVALUATION & DISCUSSION

BUG DISCOVERED During the verification of Ratatoskr Z3 produced a counterexample representing a bug in the specification–specifically that a process could override an already committed value under subtle conditions. This bug carried over into our specification from Figure 1 of Gafni and Lamport [48]. The problem is in the phase 0 recovery, allowing a processor to override an already committed, violating consistency.

From the Z3 counterexample, we deciphered a trace of actions triggering the bug, described next. Assume we have a single processor node and three disks, all zero initialized, as shown below. Recall that we denote no input using $\perp$.

		tempSet		
	dblock	1	2	3
mbal	0	0	0	0
bal	0	0	0	0
inp	$\perp$	$\perp$	$\perp$	$\perp$

	Disks		
	1	2	3
mbal	0	0	0
bal	0	0	0
inp	$\perp$	$\perp$	$\perp$

The table on the right captures the disk states, while the table on the left shows the processor state, dblock. The temporary reading set maintained by the processor, tempSet, is shown next to dblock.

At this stage, the processor has executed phase 0 recovery, assigning dblock to a value from tempSet with the largest mbal. Next, the processor node executes phase 1, using ballot number 3. Disks 2 and 3 become unavailable and only disk 1 accepts the phase 1 writes.

		tempSet		
	dblock	1	2	3
mbal	3	0	0	0
bal	0	0	0	0
inp	$\perp$	$\perp$	$\perp$	$\perp$

	Disks		
	1	2	3
mbal	3	0	0
bal	0	0	0
inp	$\perp$	$\perp$	$\perp$

Suppose now that the processor crashes, and disk 1 becomes unavailable. During recovery, the processor node reads disks 2 and 3, which constitute a quorum. Recovering the following state (here '-' represents uninitialized state):

tempSet

	dblock	1	2	3
mbal	0	-	0	0
bal	0	-	0	0
inp	$\perp$	-	$\perp$	$\perp$

Disks

	1	2	3
mbal	3	0	0
bal	0	0	0
inp	$\perp$	$\perp$	$\perp$

With disk 1 unavailable, the processor now successfully goes through phases 1 and 2 using disks 2 and 3, commiting value v with ballot number 2 (which is larger than 0, the largest mbal value in its tempSet).

tempSet

	dblock	1	2	3
mbal	2	-	0	0
bal	2	-	0	0
inp	v	-	$\perp$	$\perp$

Disks

	1	2	3
mbal	3	2	2
bal	0	2	2
inp	$\perp$	v	v

After successfully committing v, the processor node crashes, and disk 1 becomes available again. This time, during recovery, the processor recovers the following state:

tempSet

	dblock	1	2	3
mbal	3	3	2	2
bal	0	0	2	2
inp	$\perp$	$\perp$	v	v

Disks

	1	2	3
mbal	3	2	2
bal	0	2	2
inp	$\perp$	v	v

Critically, the processor updated its dblock using the values from disk 1, as it has the largest mbal value. From here, it proceeds to phase 2 committing a new value, overriding v in violation of consistency. The correct behavior is setting dblock to the block in tempSet with the largest *bal* value, instead of the largest mbal value, separately keeping track of the mbal value. The final dblock should then have the largest mbal value seen while bal and inp should be equal to the block in tempSet with the largest bal (in the example above, dblock should be $\langle 3, 2, v \rangle$).

The TLA+ specification shown in the appendix of Gafni and Lamport [48] correctly picks the block with the maximum bal value, instead of the maximum mbal value.

push-button verification of protocols Verifying distributed protocols such as Disk Paxos is non-trivial, even with a high-degree of automation. The reason is their correctness relies on a large number of invariants that need to be established, which requires manual work. Nonetheless, high degree of automation as in Ratatoskr is still valuable and counterexamples are useful for debugging. Z3 sometimes fails to come up with counterexamples when trying to prove complicated invariants with a large number of quantifiers. Temporarily reducing system parameters such as the number of processors and disks helps Z3 construct counterexamples. These counterexamples proved invaluable for debugging and helping with the discovery of missing invariants. Once the bugs have been fixed or missing invariants added, Z3 verifies the system successfully.

LIMITATION There are several limitations to Ratatoskr. For one, there is no verified implementation. Refining the protocol specification to an implementation is future work. Additionally, although it guarantees consistency of committed values, Ratatoskr does not guarantee that a value can ever be committed (i.e., Ratatoskr does not prove liveness [62]).

6.5 RELATED WORK

MACHINE CHECKED PROOFS OF CONSENSUS PROTOCOLS IronFleet [62] is a system for building and verifying distributed systems, built using the Dafny [93] verifier. Using IronFleet the authors built IronKV, a distributed key value store, and IronRSL, an implementation of Paxos with a machine checked proof that it satisfies both its safety specification as well as liveness.

Verdi [156] takes another approach to building distributed systems by use of system transformers. Verdi is built using the Coq [149] theorem prover. Verdi developers write and verify their implementation in Coq, and then apply a system transformer to the system transforming it into an equivalent implementation that is safe in a different environment. One of the system transformers Verdi supports is a Raft transformer, which adds replication for fault tolerance.

Both of these systems use higher-order and undecidable logic to specify and express complex protocol and implementation properties. Compared to Ratatoskr, they therefore require a large amount of manual effort in proof writing. IronFleet extends the verification properties to the application state machine, which neither Ratatoskr nor Verdi does. Ratatoskr has no proof of liveness.

In all cases the top level proof ensures a strict-serializble execution of commands: that the state machine acts as if it were executed sequentially on a single machine.

PROTOCOL PROOFS Jaskelioff and Merz [71] specified and proved the correctness of the Disk Paxos protocol using the Isabelle interactive theorem prover. The Isabelle specification closely matches the TLA+ specification described by Gafni and Lamport [48].

6.6 CONCLUSION

Ratatoskr is a specification of the Disk Paxos consensus protocol with a proof of correctness for consensus. Our experience shows that specifying and verifying complex protocol is feasible with push-button verification. Although coming up with the necessary invariants still requires substantial amount of manual efforts which are not be automated, counterexamples are immensely useful for discovering missing invariants. The verification of Ratatoskr uncovered a bug in the Disk Paxos algorithm as described by the original paper.

7 REFLECTION

This chapter discusses and documents experiences building the push-button verified systems outlined in this dissertation and reflects on some of the different design decisions and techniques used.

7.1 CHOICE OF MEMORY AND DATA ENCODING

Scaling push-button verification of systems requires an efficient memory model, one that is faithful to the system while generating constraints that are easy on the solver. A disparity between the memory model and the access pattern used by the implementation results in additional constraints generated due to conversion between the encodings, adding more work for the verifier. For instance, if the verifier represents memory as a one-dimensional array of bytes, then writing a 64-bit word to memory requires a total of 8 extracts and write operations, one for each byte in the word. Since verification performance is often dependent on the number of operations performed, this amplification can lead to verification bottlenecks.

We tried a number of different disk models for Yggdrasil in our initial prototype, most of which were unsuccessful as they did not scale. One such approach modeled the disk as an array of 4096-bit bitvectors, requiring a large number of extractions and concatenations, leading to timeouts in the solver. The disk model we decided on was scalable by carefully matching the implementation with minimal such conversion. A disk is a flat array of blocks, indexed by a 64-bit block index, where each block consists of 512 64-bit bitvectors. Because the implementation usually reads and writes 64-bit aligned words, this memory model is both efficient for verification and execution.

Our experience with the memory model in Yggdrasil influenced our decision to used a set of disjoint typed memory arrays in Hyperkernel, which require no data conversion for memory access since all accesses to a particular region are always of the same type. Nickel used a combination of the 64-bit field model from Yggdrasil and the typed pages from Hyperkernel.

While these memory models are efficiently solvable in practice and sufficiently expressive for our systems described in this thesis, they have several limitations. Reasoning about symbolic pointers is, in general, not supported, and all accesses within a memory region have to be of the same size (e.g., byte addressing and word addressing the same region in memory is not supported). For these reasons, the verifier is unstable and hard to work with, especially as the system size grows. Relatively simple changes to the implementation, or a change in LLVM optimizations, would break the verification by generating unsupported memory accesses, requiring a patchwork of fixes. If we were to redesign the verifier from scratch, we would instead opt for a more robust memory model, such as the one in Serval as described by Nelson et al. [116].

7.2 VALIDATION

Validation [123, 139, 144] is a technique that we used both in Yggdrasil (Chapter 3) and Hyperkernel (Chapter 4). The key idea is that instead of verifying the functional correctness of an algorithm or data structure, we instead verify a validator. The validator is much simpler to verify, and it checks the output of the algorithm. The checker halts the system if the unverified component produces an unexpected outcome. While this is safe, it produces weaker guarantees compared to full verification, effectively trading verification guarantees for verification simplicity. The distinction is that validation can guarantee that the system does not produce incorrect output; it cannot guarantee that it produces correct output.

Yggdrasil uses validation for the block allocation algorithm, directory reference counters, and directory entry lookup. All of these components either required a loop or a linked structure, making them hard to verify. Initially, we were unsure that it would be feasible to prove their correctness in a push-button fashion. Since their functional correctness was not critical to our primary goal, atomic crash safety, we found the use of validation to be an acceptable balance to strike.

In contrast, Hyperkernel makes heavy use of reference counters, and their correctness is critical to the safety of the kernel. Consequently, validation was less than ideal, and we were motivated to find a suitable encoding to verify reference count correctness efficiently. Before deciding on the encoding described in Section 4.2.3, our initial attempt to encode reference counters was a naïve inductive definition, defining `refcnt(n)` in terms of `refcnt(n-1)`. This naïve encoding worked for small memory but led to timeouts as we increased its size, making the encoding inadequate for any non-trivial system.

Similarly, both Yggdrasil and Hyperkernel use validation for page allocation. Hyperkernel allocates pages via trap handlers taking the desired page number as an input; the kernel then validates that the page number is available. This design ensures that the allocator never reuses a page that is in use but does not guarantee full utilization of all free pages. There are two issues with this design, making it unsuitable for NiStar. First, it exposes metadata to the user, which is exploitable as a communication channel. For example, process p can communicate a secret bit to process q by deciding to allocate a page or not. Then, q scans the metadata to see if p has performed the allocation, learning the secret bit. Second, even if the metadata was not exposed, and the allocator was in the kernel using validation as Yggdrasil does, the weaker validation property introduces a subtle bidirectional flow between all processes and the allocator, and therefore transitively between all processes. To see why, consider a simple off-by-one bug in the allocator, causing it never to allocate the last available page. Now, even if a process that has sufficient quota tries to allocate a page, it may get an error condition from the allocator depending on the actions of other running processes. Since NiStar can not take advantage of validation for allocation, we developed a reusable library for efficiently encoding ordered sets (implemented as a linked list) for verification, which the kernel uses to manage allocation decisions while controlling information flow.

Validation can be an effective technique to simplify verification, but it is by no means a silver bullet since the resulting guarantees are weaker compared to functional verification. If we were to re-implement Yggdrasil, we would probably rely less on validation, and instead strengthen our top-level guarantees by using similar encodings as Hyperkernel and NiStar for reference counting and linked lists.

8 CONCLUSION

Modern applications rely on systems software for their security, robustness, and reliability. This dissertation argues that by treating automated verification as a first-class design goal and co-designing it with verification, it is possible to build correct systems with a substantially reduced developer burden. In support of this, we presented four case studies, Yggdrasil, Hyperkernel, Nickel and Ratatoskr. Yggdrasil presents crash-refinement, a new definition of file system correctness that is amenable to efficient SMT solving and presents several techniques to scale up automated verification to full file systems, including stacking of abstractions and the separation of data representations. We described Hyperkernel, an OS kernel formally verified with a high degree of proof automation and low proof burden by applying the finite interface design. We also built Nickel, a framework for designing and verifying information flow control systems through automated verification techniques. Nickel helps developers eliminate covert channels from the interface design. Additionally, Nickel presents a new formulation of noninterference and the meta-theory for its unwinding strategy, decomposing a complicated noninterference specification to make it amenable to automated verification. Finally, we presented Ratatoskr, a specification and proof of correctness for the Disk Paxos algorithm in a push-button fashion.

Our experience building these systems shows that the proof burden is low and that through formal verification we can prevent common bugs found in low-level systems software. Push-button verification is a trade-off between generality and automation, and we have developed several techniques to achieve full automation for common programming patterns found in low-level systems software.

While much work remains to explore the limits of the push-button approach, we believe this is a promising direction for future system designs and that co-designing systems with proof automation provides strong guarantees while minimizing manual effort.

BIBLIOGRAPHY

[1] https://www.musl-libc.org/. 2018.

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

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

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

[5] Sidney Amani et al. "Cogent: Verifying High-Assurance File System Implementations." In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, Apr. 2016, pp. 175–188.

[6] Andrew W. Appel and Kai Li. "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, Apr. 1991, pp. 96–107.

[7] *Avionics Application Software Standard Interface: Part 1, Required Services*. Aug. 2015.

[8] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In: *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. Amsterdam, The Netherlands, Nov. 2005, pp. 364–387.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "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, Oct. 2009, pp. 29–44.

[10] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. "Cython: The Best of Both Worlds." In: *Computing in Science Engineering* 13.2 (2011). http://cython.org/, pp. 31–39.

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

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

[13] David E. Bell and Leonard La Padula. *Secure Computer System: Unified Exposition and Multics Interpretation*. Tech. rep. MTR-2997, Rev. 1. Bedford, MA: MITRE Corp., Mar. 1976.

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

[15] Kenneth J. Biba. *Integrity Considerations for Secure Computer Systems*. Tech. rep. MTR-3153. Bedford, MA: MITRE Corp., Apr. 1977.

[16] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. "The KeyKOS Nanokernel Architecture." In: *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. Apr. 1992, pp. 95–112.

[17] Jeff Bonwick. *ZFS: The Last Word in Filesystems*. https://blogs.oracle.com/bonwick/entry/zfs_the_last_word_in. Oct. 2005.

[18] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. "Specifying and Checking File System Crash-Consistency Models." In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, Apr. 2016, pp. 83–98.

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018, pp. 991–1008.

[20] *CWE-209: Information Exposure Through an Error Message*. `https://cwe.mitre.org/data/definitions/209.html`. Jan. 2018.

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

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

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

[24] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. "Abstractions for Usable Information Flow Control in Aeolus." In: *Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, June 2012.

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

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

[27] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver." In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Rome, Italy, Mar. 2013, pp. 93–107.

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

[29] Jonathan Corbet. *Thoughts on the ext4 panic*. `https://lwn.net/Articles/521803/`. Oct. 2012.

[30] Jonathan Corbet. *A tale of two data-corruption bugs*. https://lwn.net/Articles/645720/. May 2015.

[31] David Costanzo, Zhong Shao, and Ronghui Gu. "End-to-End Verification of Information-Flow Security for C and Assembly Programs." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, June 2016, pp. 648–664.

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

[33] Christoffer Dall and Jason Nieh. "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, Mar. 2014, pp. 333–347.

[34] Dorothy E. Denning. "A Lattice Model of Secure Information Flow." In: *Communications of the ACM* 19.5 (May 1976), pp. 236–243.

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

[36] Petros Efstathopoulos and Eddie Kohler. "Manageable Fine-Grained Information Flow." In: *Proceedings of the 3rd ACM EuroSys Conference*. Glasgow, Scotland, Apr. 2008, pp. 301–313.

[37] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. "Labels and Event Processes in the Asbestos Operating System." In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. Brighton, United Kingdom, Oct. 2005, pp. 17–30.

[38] Sebastian Eggert. "Security via Noninterference: Analyzing Information Flows." PhD thesis. Kiel University, July 2014.

[39] Sebastian Eggert and Ron van der Meyden. "Dynamic intransitive noninterference revisited." In: *Formal Aspects of Computing* 29.6 (June 2017), pp. 1087–1120.

[40] Kevin Elphinstone and Gernot Heiser. "From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels?" In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, Nov. 2013, pp. 133–150.

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

[42] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. "Inside a Verified Flash File System: Transactions & Garbage Collection." In: *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments*. San Francisco, CA, July 2015.

[43] FUSE. *Filesystem in Userspace*. https://github.com/libfuse/libfuse. 2016.

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

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

[46] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis." In: *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China, Apr. 2017, pp. 555–568.

[47] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. "HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security." In: *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada, Oct. 2018.

[48] Eli Gafni and Leslie Lamport. "Disk paxos." In: *Distributed Computing* 16.1 (2003), pp. 1–20.

[49] Gregory R. Ganger and Yale N. Patt. "Metadata Update Performance in File Systems." In: *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Monterey, CA, Nov. 1994, pp. 49–60.

[50] Qian Ge, Yuval Yarom, and Gernot Heiser. "No Security Without Time Protection: We Need a New Hardware-Software Contract." In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. Jeju Island, South Korea, Aug. 2018.

[51] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. "Hails: Protecting Data Privacy in Untrusted Web Applications." In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, Oct. 2012.

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

[53] John Graham-Cumming and J. W. Sanders. “On the Refinement of Non-interference.” In: *Proceedings of the Computer Security Foundations Workshop VI*. Franconia, NH, June 1991, pp. 35–42.

[54] Ronghui Gu. *Private communication*. Aug. 2018.

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

[56] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. “R2: An Application-Level Kernel for Record and Replay.” In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, Dec. 2008, pp. 193–208.

[57] Robert Hagmann. “Reimplementing the Cedar File System Using Logging and Group Commit.” In: *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*. Austin, TX, Nov. 1987, pp. 155–162.

[58] J. T. Haigh and W. D. Young. “Extending the Noninterference Version of MLS for SAT.” In: *IEEE Transactions on Software Engineering* SE-13.2 (Feb. 1987), pp. 141–150.

[59] William R. Harris, Somesh Jha, and Thomas Reps. “DIFC Programs by Automatic Instrumentation.” In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL, Oct. 2010, pp. 284–296.

[60] William R. Harris, Somesh Jha, Thomas Reps, and Sanjit A. Seshia. “Program Synthesis for Interactive-Security Systems.” In: *Formal Methods in System Design* 51.2 (Nov. 2017), pp. 362–394.

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

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

[63] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. "Reducing fsck time for ext2 file systems." In: *Proceedings of the Linux Symposium*. Ottawa, Canada, June 2006, pp. 395–408.

[64] Valerie Henson. *The many faces of fsck*. https://lwn.net/Articles/248180/. Sept. 2007.

[65] C. A. R. Hoare. "Proof of Correctness of Data Representations." In: *Acta Informatica* 1.4 (Dec. 1972), pp. 271–281.

[66] C. A. R. Hoare. "Communicating Sequential Processes." In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677.

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

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

[69] Suman Jana and Vitaly Shmatikov. "Memento: Learning Secrets from Process Footprints." In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2012, pp. 143–157.

[70] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. "Establishing Browser Security Guarantees through Formal Shim Verification." In: *Proceedings of the 21st USENIX Security Symposium*. Bellevue, WA, Aug. 2012, pp. 113–128.

[71] Mauro Jaskelioff and Stephan Merz. "Proving the correctness of disk paxos." In: *Archive of Formal Proofs* 2005 (2005).

[72] Dylan Johnson. *Porting Hyperkernel to the ARM Architecture*. Tech. rep. UW-CSE-17-08-02. University of Washington, Aug. 2017.

[73] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. "DFS: A File System for Virtualized Flash Storage." In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA, Feb. 2010, pp. 1–15.

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

[75] 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. "Application Performance and Flexibility on Exokernel Systems." In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France, Oct. 1997, pp. 52–65.

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

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

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

[79] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive formal verification of an OS microkernel." In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), 2:1–70.

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

[81] Rafal Kolanski. "Verification of Programs in Virtual Memory Using Separation Logic." PhD thesis. University of New South Wales, July 2011.

[82] Eric Koskinen and Junfeng Yang. "Reducing Crash Recoverability to Reachability." In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, Jan. 2016, pp. 97–108.

[83] Maxwell Krohn and Eran Tromer. "Noninterference for a Practical DIFC-Based Operating System." In: *Proceedings of the 30th IEEE Symposium on Security and Privacy*. Oakland, CA, May 2009, pp. 61–76.

[84] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information Flow Control for Standard OS Abstractions." In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, WA, Oct. 2007, pp. 321–334.

[85] LTP. *Linux Test Project*. `http://linux-test-project.github.io/`. 2016.

[86] Leslie Lamport. "The Part-Time Parliament." In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

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

[88] Leslie Lamport and Clocks Time. "the Ordering of Events in a Distributed System." In: *Communications of the ACM* 21.7 (1979), p. 558.

[89] Butler W. Lampson. "A Note on the Confinement Problem." In: *Communications of the ACM* 16.10 (Oct. 1973), pp. 613–615.

[90] Butler W. Lampson. “Hints for Computer System Design.” In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*. Bretton Woods, NH, Oct. 1983, pp. 33–48.

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

[92] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. “Taming Undefined Behavior in LLVM.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, June 2017, pp. 633–647.

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

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

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

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

[97] John Lions. *Lions’ Commentary on Unix*. 6th. Peer-to-Peer Communications, Aug. 1996.

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

[99] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. “Provably Correct Peephole Optimizations with Alive.” In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, June 2015, pp. 22–32.

[100] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. “A Study of Linux File System Evolution.” In: *ACM Transactions on Storage (TOS)* 10.1 (Jan. 2014), pp. 31–44.

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

[102] Heiko Mantel. "Preserving Information Flow Properties under Refinement." In: *Proceedings of the 22nd IEEE Symposium on Security and Privacy*. Oakland, CA, May 2001, pp. 78–91.

[103] Marshall Kirk McKusick. "Journaled Soft-updates." In: *BSDCan*. Ottawa, Canada, May 2010.

[104] Marshall Kirk McKusick and T. J. Kowalski. "Fsck—The UNIX File System Check Program." In: *UNIX System Manager's Manual (SMM), 4.4 Berkeley Software Distribution*. University of California, Berkeley, Oct. 1996.

[105] Ron van der Meyden. "Architectural Refinement and Notions of Intransitive Noninterference." In: *Formal Aspects of Computing* 24.4–6 (July 2012), pp. 769–792.

[106] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." In: *ACM Transactions on Database Systems* 17.1 (Mar. 1992), pp. 94–162.

[107] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. "CrashMonkey and ACE: Systematically Testing File-System Crash Consistency." In: *ACM Transactions on Storage (TOS)* 15.2 (2019), pp. 1–34.

[108] Bodo Möller. *Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures*. https://www.openssl.org/~bodo/tls-cbc.txt. Sept. 2014.

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

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

[111] Toby Murray, Robert Sison, and Kai Engelhardt. "Covern: A Logic for Compositional Verification of Information Flow Control." In: *Proceedings of the 3rd IEEE European Symposium on Security and Privacy*. London, United Kingdom, Apr. 2018, pp. 16–30.

[112] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. "Noninterference for Operating System Kernels." In: *Proceedings of the 2nd International Conference on Certified Programs and Proofs*. Kyoto, Japan, Dec. 2012, pp. 126–142.

[113] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. "seL4: from General Purpose to a Proof of Information Flow Enforcement." In: *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA, May 2013, pp. 415–429.

[114] Andrew C. Myers and Barbara Liskov. "Protecting Privacy using the Decentralized Label Model." In: *ACM Transactions on Computer Systems* 9.4 (Oct. 2000), pp. 410–442.

[115] Andrew Myers and Barbara Liskov. "A Decentralized Model for Information Flow Control." In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France, Oct. 1997, pp. 129–147.

[116] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Serval: scaling symbolic evaluation for automated verification of systems code." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, Oct. 2019.

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

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

[119] David von Oheimb. "Information Flow Control Revisited: Noninfluence = Noninterference + Nonleakage." In: *Proceedings of the 9th European Symposium on Research in Computer Security*. Sophia Antipolis, France, Sept. 2004, pp. 225–243.

[120] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "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, Mar. 2011, pp. 305–318.

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

[122] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications." In: *Proceedings of the 11th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI). Broomfield, CO, Oct. 2014, pp. 433–448.

[123] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation." In: *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lisbon, Portugal, 1998, pp. 151–166.

[124] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Model-Based Failure Analysis of Journaling File Systems." In: *Proceedings of the 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Yokohama, Japan, 2005, pp. 802–811.

[125] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "IRON File Systems." In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. Brighton, United Kingdom, Oct. 2005, pp. 206–220.

[126] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and P. Madhusudan. "Natural Proofs for Structure, Data, and Separation." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, June 2013, pp. 16–19.

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

[128] Eric Reed. *Patina: A Formalization of the Rust Programming Language*. Tech. rep. UW-CSE-15-03-02. University of Washington, Feb. 2015.

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

[130] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. Copenhagen, Denmark, July 2002, pp. 55–74.

[131] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. "Automating Formal Proofs for Reactive Systems." In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014, pp. 452–462.

[132] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. "SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems." In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 38–53.

[133] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-Tree Filesystem." In: *ACM Transactions on Storage (TOS)* 9.3 (Aug. 2013), 9:1–32.

[134] M. Rosenblum and J. Ousterhout. "The Design and Implementation of a Log-Structured File System." In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, CA, Oct. 1991, pp. 1–15.

[135] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. "Laminar: Practical Fine-Grained Decentralized Information Flow Control." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, June 2009.

[136] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. "Error Propagation Analysis for File Systems." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland, June 2009, pp. 270–280.

[137] John Rushby. "Design and Verification of Secure Systems." In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, CA, Dec. 1981, pp. 12–21.

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

[139] Hanan Samet. "Proving the Correctness of Heuristically Optimized Code." In: *Communications of the ACM* 21.7 (July 1978), pp. 570–582.

[140] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger Vernon Austel, and David Toll. "Verification of a Formal Security Model for Multiapplicative Smart Cards." In: *Proceedings of the 6th European Symposium on Research in Computer Security*. Toulouse, France, Oct. 2000, pp. 17–36.

[141] Gerhard Schellhorn, Gidon Ernst, Jorg Pfähler, Dominik Haneberg, and Wolfgang Reif. "Development of a Verified Flash File System." In: *Proceedings of the ABZ Conference*. June 2014.

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

[143] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial." In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

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

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

[146] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. "Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems." In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Copenhagen, Denmark, Sept. 2012, pp. 201–214.

[147] Zachary Tatlock and Sorin Lerner. "Bringing Extensibility to Verified Compilers." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Canada, June 2010, pp. 111–121.

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

[149] The Coq Development Team. *The Coq Proof Assistant, version 8.9.0*. Jan. 2019. DOI: `10.5281/zenodo.2554024`. URL: `https://doi.org/10.5281/zenodo.2554024`.

[150] Chandramohan A. Thekkath and Henry M. Levy. "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, Oct. 1994, pp. 110–119.

[151] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages." In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014, pp. 530–541.

[152] Linus Torvalds. *Re: [patch] measurements, numbers about `CONFIG_OPTIMIZE_INLINING=y` impact*. https://lkml.org/lkml/2009/1/9/497. Jan. 2009.

[153] Ta-chung Tsai, Alejandro Russo, and John Hughes. "A Library for Secure Multi-threaded Information Flow in Haskell." In: *Proceedings of the 20th IEEE Computer Security Foundations Symposium*. Venice, Italy, July 2007, pp. 187–202.

[154] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. "Specification and Verification of the UCLA Unix Security Kernel." In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 118–131.

[155] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. "SpaceSearch: A Library for Building and Verifying Solver-Aided Tools." In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Sept. 2017.

[156] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, June 2015, pp. 357–368.

[157] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. "Efficiently Solving Quantified Bit-Vector Formulas." In: *Proceedings of the 10th Conference on Formal Methods in Computer-Aided Design*. Lugano, Switzerland, Oct. 2010, pp. 239–246.

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

[159] Jean Yang and Chris Hawblitzel. "Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Canada, June 2010, pp. 99–110.

[160] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. "Using Model Checking to Find Serious File System Errors." In: *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA, Dec. 2004, pp. 273–287.

[161] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. "Automatically Generating Malicious Disks using Symbolic Execution." In: *Proceedings of the 27th IEEE Symposium on Security and Privacy*. Oakland, CA, May 2006, pp. 243–257.

[162] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. "eXplode: A Lightweight, General System for Finding Serious Storage System Errors." In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Nov. 2006, pp. 131–146.

[163] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems." In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, Oct. 2014, pp. 249–265.

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

[165] Steve Zdancewic and Andrew C. Myers. "Robust Declassification." In: *Proceedings of the 14th IEEE workshop on Computer Security Foundations*. Cape Breton, Canada, June 2001.

[166] Karen Zee, Viktor Kuncak, and Martin C. Rinard. "Full Functional Verification of Linked Data Structures." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Tucson, AZ, June 2008, pp. 349–361.

[167] Nickolai Zeldovich. *Private communication*. Apr. 2018.

[168] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control." In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, CA, Apr. 2008, pp. 293–308.

[169] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar." In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, Nov. 2006, pp. 263–278.

[170] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar." In: *Communications of the ACM* 54.11 (Nov. 2011), pp. 93–101.

[171] Kehuan Zhang and XiaoFeng Wang. “Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems.” In: *Proceedings of the 18th USENIX Security Symposium*. Montreal, Canada, Aug. 2009, pp. 17–32.

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

[173] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. “Reasoning About Information Flow Security of Separation Kernels with Channel-based Communication.” In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Eindhoven, The Netherlands, Apr. 2016, pp. 791–810.

[174] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. “Torturing Databases for Fun and Profit.” In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, Oct. 2014, pp. 449–464.

[175] *`dup`, `dup2`, `dup3` - duplicate a file descriptor*. `http://man7.org/linux/man-pages/man2/dup.2.html`. Dec. 2016.