Verus: A Practical Foundation for Systems Verification

SOSP Accepted Paper #125 (Draft in Shepherding)

Andrea Lattuada*
MPI-SWS
Saarbrücken, Germany

Travis Hance
Carnegie Mellon University
Pittsburgh, PA, United States

Jay Bosamiya†
Microsoft Research
Redmond, WA, United States

Matthias Brun
ETH Zurich
Zürich, Switzerland

Chanhee Cho
Carnegie Mellon University
Pittsburgh, PA, United States

Hayley LeBlanc
University of Texas at Austin
Austin, TX, United States

Pranav Srinivasan
University of Michigan
Ann Arbor, MI, United States

Reto Achermann
University of British Columbia
Vancouver, BC, Canada

Tej Chajed
University of Wisconsin-Madison
Madison, WI, United States

Chris Hawblitzel
Microsoft Research
Redmond, WA, United States

Jon Howell
VMware Research
Bellevue, WA, United States

Jacob R. Lorch
Microsoft Research
Redmond, WA, United States

Oded Padon*
Weizmann Institute of Science
Rehovot, Israel

Bryan Parno
Carnegie Mellon University
Pittsburgh, PA, United States

Abstract

Formal verification is a promising approach to eliminate bugs at compile time, before they ship. Indeed, our community has verified a wide variety of system software. However, much of this success has required heroic developer effort, relied on bespoke logics for individual domains, or sacrificed expressiveness for powerful proof automation.

Building on prior work on Verus, we aim to enable faster, cheaper verification of rich properties for realistic systems. We do so by integrating and optimizing the best choices from prior systems, tuning our design to overcome barriers encountered in those systems, and introducing novel techniques.

We evaluate Verus's effectiveness with a wide variety of case-study systems including distributed systems, an OS page table, a library for NUMA-aware concurrent data structure replication, a crash-safe storage system, and a concurrent memory allocator, together comprising 6.1K lines of implementation and 31K lines of proof. Verus verifies code 3–61x faster and with less effort than the state of the art.

Our results suggest that Verus offers a platform for exploring the next frontiers in system verification research. Because Verus builds on Rust, Verus is also positioned for wider use in production by developers who have already adopted Rust in the pursuit of more robust systems.

1 Introduction

Society increasingly counts on the correctness, reliability, and security of system software, i.e., fundamental software infrastructure that includes file systems, operating systems, databases, memory allocators, and libraries for cryptography and distributed protocols. Such software is often inherently low-level (e.g., manipulating raw bytes, interfacing directly with devices, or operating without a garbage collector), since higher-level software depends on it for foundational abstractions, like unlimited virtual memory or reliable operation. System software must also hit stringent performance targets since it sits on the critical path for the software above it. However, performance optimizations, especially those involving concurrency, add complexity. Unsurprisingly, these systems are notoriously difficult to get right.

Formally verifying software is a promising approach for ensuring its correctness and reliability. Indeed, our community has seen a series of successful demonstrations verifying a wide variety of system software (§5). However, much of this success has required heroic developer effort, relied on bespoke logics for individual domains (e.g., crash safety [1–3]), or sacrificed expressiveness for powerful automation [1, 4–9].

In our work, we aim to enable faster, cheaper verification of rich properties for realistic systems. We do so by integrating and optimizing the best choices from prior systems, tuning our design to overcome barriers they encountered, and introducing novel techniques to simplify concurrency reasoning.

Our work builds on Verus [10], which uses Rust's type system [11, 12] to manage resources and verify `unsafe` Rust code. While that early work focuses on query encoding and a

*Work done while at VMware Research.
†Work done while at Carnegie Mellon University.

formalization thereof, we see Verus as an exciting framework for system verification. Here, we describe how we extend it with system-relevant features, evaluating them in systems code. Our ambition is two-fold: first, to make Verus a foundation for exploring the next frontier in system verification research, and second, to open system verification to a broad community of developers without deep verification expertise.

Core to Verus is the acknowledgment that verifying system software entails many kinds of reasoning at many different levels of abstraction (§2), from low-level details of memory access and bit manipulation, to the high-level challenges of defining and proving that a file system is crash-safe or that a distributed system achieves consensus. Rather than tackle this complexity with a single generic solution, Verus carefully organizes proof obligations in a way that plays to the strengths of various proof automation strategies, and hence minimizes developer effort.

Across all proof levels, Verus integrates two powerful reasoning techniques. By default, it provides aggressively optimized versions of general-purpose, semi-automated proof techniques (§3.1) that have shown success in one line of verified systems [13–18]. For portions of the system that can be described in EPR [19], a restricted logic, Verus shows how to soundly integrate another line of work [7, 8, 20–24] so proofs of these portions are fully automated (§3.2).

Verus also includes custom automation for reasoning challenges at specific levels in real systems. For example, certain coding idioms like bit manipulation and nonlinear arithmetic confound prior tools; our Verus extensions add dedicated support for such idioms (§3.3). Real systems also exploit shared-memory concurrency, so we extend Verus (§3.4) to allow a developer to describe a plan for sharding state across threads, show that threads locally obey the plan, and create invariants about the global program state from the plan.

Evaluating a new general-purpose system verification framework is challenging, given that entire papers have been devoted to verifying a single system. Hence we adopt a multi-level evaluation. We start with a set of verification "millibenchmarks": programs we verify in Verus and in five comparable tools. This helps us assess the impact of Verus's design decisions on verification performance in examples small enough to analyze in some detail.

We then show that the benefits seen at small scale extend to larger systems by porting the specifications, code, and proofs from several prior large-scale verification efforts to Verus and comparing the developer experience in terms of verification time and developer effort. These case studies include a distributed system client and a library that creates a linearizable NUMA-aware concurrent data structure from a black-box sequential one.

Of course, starting from correct code and successful proofs is overly optimistic, so we also evaluate similar metrics as we verify three new system components from scratch, including an OS page table, a concurrent memory allocator, and a storage system designed for production. While our results are positive,[1] the ultimate evaluation will come from future projects using our extended Verus to verify systems at new levels of scale and complexity. Indeed, other research groups have already used Verus to produce verified cluster-management controllers [25], a verified microkernel [26], a security module for confidential VMs [27], and for LLM-based code production [28].

Any verification result is limited by its TCB. Verus's results depend on the correctness of the top-level specifications, the Verus verifier, the solvers it relies on, and the Rust compiler.

In summary, we:

- Present the systems-relevant aspects of Verus's design (§3), unifying prior approaches in a single framework and simplifying system proofs via novel techniques.
- Show how to provide the proof-free automation achieved by prior work [7, 8, 20–24] without sacrificing expressiveness or developer freedom.
- Describe Verus's new approach to allowing the developer to reason about concurrent execution *at the level of the application*, rather than the low-level mathematical objects required by prior work [3, 29, 30]
- Evaluate Verus as a system verification language through millibenchmarks and five case studies, producing over 6.1K lines of implementation and 31K lines of proof. These case studies show how Verus's features support verifying a variety of systems for correctness and reliability. Verus also produces verification results *orders of magnitude* faster than prior automated tools.

2 Verus Overview: Verification for Systems

When constructing a complex system, developers typically tame the complexity by thinking about the system at different levels of abstraction. These range from low-level concerns like memory safety to high-level concerns like crash safety. Each level is challenging, but the ultimate challenge is ensuring that all of the levels together produce a correct system.

As detailed in Figure 1, Verus helps developers with every aspect of this reasoning. Below, we provide an overview of Verus's key features. Given space constraints, the subsequent sections then provide more details on a selection of those features. §4 evaluates the overall impact on system verification.

Cross-Layer Proof Automation. At the scale of complexity in modern systems, encoding both the code and the abstractions needed to reason about it in a mechanized form, if done naively, can lead to huge proof obligations. Prior work (§5) tries to tame the complexity of proving correctness by sacrificing either automation or expressiveness. Verus instead combines the best of both worlds, providing aggressively optimized, general-purpose, semi-automated proof techniques

[1] Verus is open source at https://github.com/verus-lang/verus and all of our case studies are open source at https://verus-lang.github.io/paper-sosp24-artifact/case-studies.

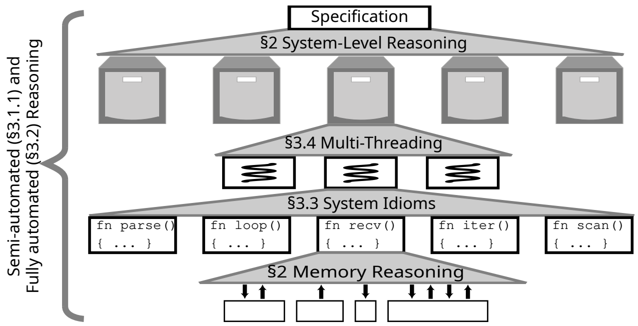


Figure 1. Verus Overview. *Verus offers powerful automated and semi-automated reasoning techniques that apply to the full system stack, as well as reasoning techniques tuned to specific stack levels.*

(§3.1) that improve on prior widely-used tools, plus full proof automation for special cases (§3.2).

Verus also includes reasoning techniques tuned for particular levels in the stack, smoothing out aspects of system verification that prior work has struggled with.

Memory Reasoning. Rather than proving the memory safety of system code via an expensive combination of manual developer effort [31–33] and/or complex encodings [13, 34, 35], Verus stands on the shoulders of the Rust [11, 12] community's extensive work on ensuring safety through a fast, deterministic type checker. Rust's *ownership*-based type checker enforces that only one variable has exclusive ownership of any object at a time. Rust's growing developer population and large-scale projects in the Linux kernel [36], at Amazon [37], and at Google [38], validate that developers can use Rust's types to verify memory safety for real systems.

Rust includes an "escape hatch" that allows developers to write explicitly labeled `unsafe` code, wherein the developer accepts the obligation to ensure the code's safety, which challenges even experts [39]. Fortunately, prior work [10] shows how developers can use Verus to verify `unsafe` Rust.

Hence, Verus gives programmers C-like freedom to manage and manipulate memory, and yet it automatically proves memory safety of almost all such code.

System Idioms. At the next level, Verus includes reasoning techniques (§3.3) specifically designed for thorny system idioms that the community's experiences indicate are challenging to generically automate. These include bit manipulation [14, 16, 40] and nonlinear arithmetic [14, 41, 42].

Multi-Threading. Higher up the stack, many systems rely on concurrency for good performance, via local multi-threaded execution. Concurrent execution makes verification significantly more complex and automation challenging, since both the verifier and the developer must now consider how each thread interacts with every other thread.

Verus includes VerusSync (§3.4), a domain-specific language for reasoning about multi-threaded code directly in terms of the application's logic, ultimately producing an abstraction of the application that encapsulates the complexity of threads, locks, and other synchronization concerns.

```
verus! {
 pub struct LinkedList<V> {
   head: Option<Box<Node<V>>>,
 }

 impl<V: VerusClone> LinkedList<V> {
  pub spec fn view(self) -> Seq<V> ... { ... }
  ...
  pub fn pop(&mut self) -> (res: V)
   requires old(self).view().len() > 0,
   ensures res == old(self).view()[0] &&
     self.view() == old(self).view().skip(1),
  {
   let h = self.head.take().unwrap();
   self.head = h.next;
   assert(self.view() =~= old(self).view().skip(1));
   h.v
  }
  ...
 }
} // verus!
```

Figure 2. Verus Example. `pop` *removes the first element of a linked list and returns it. Its* **`requires`** *says that* `pop` *can only be called if the list is non-empty, and the* **`ensures`** *says that the returned result is the list's first value, and that it is removed from the list.*

System-Level Reasoning. At the top level, developers often wish to prove more than the correctness of a program in isolation; they want richer properties of the program executing in a larger context. For example, a distributed system may run several instances of the program; we want the whole system to be reliable. A storage program interacts asynchronously with an external disk; we want the system to be correct even as the program crashes and restarts.

Extensive prior work [14, 15, 18, 29, 40] has demonstrated the generality and effectiveness of modeling this system-level reasoning in terms of atomic state machines [43] that capture system state and how it evolves, so Verus supports such reasoning as a special case of VerusSync.

3 Key Aspects of Verus's Design for Systems

Here we highlight several key ideas and design decisions that enable Verus to support the verification of complex systems.

3.1 Streamlined, General-Purpose Automation

Motivation. Some of the largest verified system implementations [13–18] have relied on semi-automated *program verifiers* like VCC [13], Dafny [34], or F* [35]. Compared to *theorem provers* [31, 32], these tools are explicitly designed to verify programs as opposed to general math theorems. While they sacrifice expressivity compared with theorem provers, they still include general-purpose reasoning techniques that allow them to be adapted to different settings. Most importantly, they provide significant automation "out of the box". However, existing program verifiers have struggled in the face of modern system complexity, often forcing developers to break their code into unnaturally small units and/or to tolerate long code-prove cycles.

Our Approach. Verus's *default mode*, used in the majority of our proofs, is careful to utilize its underlying SMT solver as efficiently as possible. To illustrate, Figure 2 shows a simple example of verifiable code written in Verus. The `verus!`

macro extends Rust with annotations to guide verification. For example, the **`requires`** and **`ensures`** annotations introduce pre- and postconditions that specify correctness for executable Rust functions like `pop`. These specs refer to auxiliary `spec` functions, like the `view` function that abstracts a concrete `LinkedList` implementation as a mathematical `Seq`. Internally, Verus employs standard Floyd-Hoare logic [44, 45] to convert proof obligations, such as `pop`'s postcondition into verification conditions to send to an SMT solver. In our case, we query Z3 [46], which automatically determines whether the formula holds for all possible inputs to the function.

The approach of converting proof obligations to verification conditions is standard in many verification frameworks, but the details of language design and implementation have a big effect on the efficiency of the SMT queries. We summarize some of Verus's key optimizations below. The net effect is that Verus sends queries that are simpler and smaller *by orders of magnitude* to the SMT solver, which translates to significantly faster verification (Figure 8), so for a given verification problem, Verus can provide a more efficient code-prove cycle. Or conversely, given the same time budget as an existing tool, Verus can tackle larger verification problems.

- Unlike specification functions in Dafny [34] and F* [35], Verus spec functions are pure, total mathematical functions with no preconditions or (direct) access to the heap. Thus they can be directly encoded as SMT functions without the additional baggage of prior tools [34, 35]. As prior work explains, Rust's type system means that explicit heap-based reasoning is rarely needed, and when it *is* needed, it can be done through *ghost memory permissions* [10].
- Verus isolates reasoning about memory, bit vectors, and nonlinear arithmetic from the main SMT queries, avoiding complex interactions between different kinds of reasoning in the SMT solver and speeding up the SMT queries. We discuss some of these ideas in §3.3.
- When verifying each module in a given project, Verus aggressively prunes the context sent to the SMT solver to eliminate imported but unreachable definitions.
- To avoid excess instantiations of quantifiers (∀ and ∃) in the SMT solver, Verus treats quantifiers more conservatively than tools like Dafny [34] and F* [35] (see below).

Quantifiers and SMT solving. Typical systems verification projects use ∀ and ∃ quantifiers, for example to specify all elements of an array or all members of a table. Unfortunately, SMT solving with quantifiers is undecidable in general, so SMT solvers use heuristics to decide how to instantiate ∀ quantifiers and how to prove ∃ quantifiers. Most large verification projects based on SMT solvers use an SMT heuristic based on "triggers" [47], in which the SMT solver pattern matches on expressions appearing in the formulas to determine how to instantiate quantifiers, for example, instantiating `(∀|x: u64| ...f(x)...)` with `x` = 3 if the expression `f(3)` arises during the proof search. Here, the expression `f(x)` is called the *trigger* (or *pattern*) for the quantifier.

Verification tools, including Verus, automatically select appropriate expressions as triggers. However, the choice of triggers has a large impact on verification performance. Prior tools, like Dafny, default to selecting broad triggers that match many expressions, leading to many quantifier instantiations; in principle this creates more opportunities for the solver to automatically complete a proof, but in practice, for large systems projects, too many instantiations slow the SMT solver to the point where it times out and fails to complete the proof.

Therefore, Verus uses a more cautious policy that selects as few triggers as possible. If Verus is uncertain about the best trigger, Verus encourages the user to review the selected triggers and to consider overriding Verus's default selection with the user's choice. The result is more initial effort for users writing small projects, but better scalability to large systems projects.

3.2 Selective Use of EPR for Full Automation

Motivation. Several prior works [1, 5–8] explicitly restrict the expressivity of system properties and implementations in order to obtain powerful proof automation. The developer need only provide a little proof support, leading to remarkably low proof-to-code ratios, e.g., Hyperkernel reports 0.03:1 [4]. The Ivy verification tool [7, 8] — built on EPR (effectively propositional) logic [19] — demonstrates that despite reduced expressivity, diverse system designs and implementations can be encoded [9, 20, 48], yielding fully automated proofs and even enabling invariant inference [21–24, 49].

While EPR leads to impressive proof automation, some system components can be difficult to capture in EPR. EPR's first-order logic admits only Boolean operators, quantifiers, and uninterpreted functions. For example, EPR can express the property that a node sends at most one message per epoch as $\forall m_1, m_2.\, \mathrm{sender}(m_1) = \mathrm{sender}(m_2) \wedge \mathrm{epoch}(m_1) = \mathrm{epoch}(m_2) \rightarrow m_1 = m_2$. EPR cannot directly express many common concepts such as sequences and natural numbers; e.g., $\mathrm{epoch}(m_2) = 1 + \mathrm{epoch}(m_1)$ is disallowed. Instead, natural numbers are typically abstracted as a totally ordered set without arithmetic operations. EPR further requires that functions and quantifier-alternations are *acyclic* [20]; e.g., the sender function above that maps messages to nodes precludes both a function from nodes to messages and a property like $\forall n, e.\, e < e_{max} \rightarrow \exists m.\, \mathrm{sender}(m) = n \wedge \mathrm{epoch}(m) = e$.

Our Approach. Rather than accepting Ivy's all-or-nothing trade-off between expressivity and automation, Verus soundly integrates EPR proofs with Verus's general-purpose semi-automation; we use the latter for most parts of the system, and apply EPR to those proofs it best fits.

To use EPR in Verus, a developer **(a)** defines a protocol or a data structure without restrictions, **(b)** abstracts the protocol or data structure into EPR, **(c)** uses Verus's integrated EPR solver for automatic, predictable proofs over the abstraction,

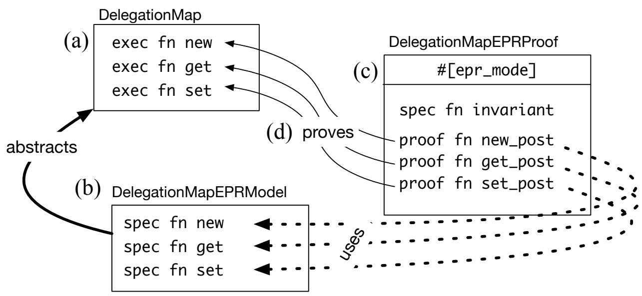


Figure 3. Verifying the IronKV Delegation Map via EPR Mode. *The concrete implementation module **(a)** is abstracted to EPR **(b)** to enable a fully-automated proof **(c)** of its invariants, which are used **(d)** to prove the implementation's correctness.*

and **(d)** exports those proof results to the original definition where they support postconditions and lemmas.

The connection between **(a)** and **(b)** is checked using Verus's default-mode automation, and in our experience is easily discharged. Step **(c)** typically involves a set of complex invariants that would have required significant manual proof in default mode, but which Verus's *EPR mode* resolves automatically. The connection **(d)** back to the original proof obligations for **(a)** is again easily checked in default mode.

We illustrate this process on the *delegation map* data structure from one of our case studies—the Verus port of IronKV (§4.2.1). Originally written in Dafny for IronFleet [15], IronKV is a distributed key-value store where each node maintains a delegation map to associate each possible key to the host responsible for it. For efficiency, the domain is stored as a compact list of pivots that represent the key ranges. The delegation map supports three operations:

`new` creates a map with all keys mapped to one host;

`set` maps a range of keys to a new host;

`get` retrieves the host responsible for a given key.

Figure 3 illustrates the components of the delegation map and its EPR proof. First, **(a)** our `DelegationMap` data-structure implements the `new`, `get`, and `set` operations in regular Rust code with pre- and post-conditions expressed in an unrestricted manner, similar to the Dafny original. Our **(b)** `DelegationMapEPRModel` abstracts the data structure and its operations into EPR. We then **(c)** write an inductive invariant in EPR—`DelegationMapEPRProof`—to automatically prove an EPR version of the postconditions of each operation. Finally, we **(d)** use these proofs to discharge the implementation's proof obligations for `new`, `get`, and `set`.

The efficient list-of-pivots implementation leads to many tricky corner cases, resulting in an extensive proof in the original Dafny version. Our initial default-mode Verus proof was also extensive; e.g., the `set` operation required ~ 300 lines of proof (mostly case splitting and assertions that instantiate quantifiers) and took several days of developer effort. However, by abstracting the keys as a totally ordered set, the proof can be expressed in Verus's EPR mode, yielding much greater proof automation. The developer still writes an inductive invariant, but the invariant is checked completely automatically, similar to Ivy [7]. However, unlike in Ivy, the implementation's pre- and post-conditions are expressed outside of EPR, seamlessly integrating with the rest of IronKV.

At a technical level, the Verus developer marks individual modules with the `#[epr_mode]` attribute, instructing Verus to check that the module's proof obligations are all within EPR. Verus confirms that structs have private fields so they act as uninterpreted types. It also checks acyclicity of the quantifier-alternation graph [20]. Finally, Verus enables Z3's *model-based quantifier instantiation*, a complete (and often fast) decision procedure for EPR. Due to technicalities of Verus's SMT encoding of polymorphic types, `epr_mode` queries are not strictly in EPR, but we have not observed any resulting incompleteness.

Our integration of EPR reasoning with a general-purpose program verifier is enabled in part by Verus's emphasis on query economy (§3.1). Most program verifiers [13, 34, 35] produce complex SMT queries even for simple programs, immediately pushing them beyond EPR. In contrast, `epr_mode`'s SMT encoding is quite close to Verus's default mode.

Summarizing, for modules that fit in EPR, Verus provides full automation, which prior work shows reduces developer effort by eliminating hundreds of lines of difficult proofs. Since Verus soundly embeds EPR in a general-purpose verifier, developing in a restricted logic is no longer an all-or-nothing choice; the developer can blend proof styles as needed.

3.3 Custom Proof Automation for System Idioms

Motivation. To gain performance, systems employ various "tricks" that are challenging to automate using generic tools.

Our Approach. Rather than feed every verification problem to a generalized SMT solver, Verus includes various (trusted) automation tools, intuitively exposed to developers.

Non-linear Arithmetic Reasoning Systems often employ non-linear arithmetic, e.g., to relate adjacent page table entries (§4.2.3). Verus automates such reasoning in two ways.

First, while non-linear arithmetic is *generally* undecidable, sub-classes are both decidable and fast in practice. Inspired by others [50, 51], Verus supports fully automated proofs of integer ring "congruence relations"—equalities built from $+$, $-$, $\times$, %, and constant exponentiation—via annotation:

```
pub proof fn subtract_mod_eq_zero(a:int, b:int, c:int)
 by(integer_ring)
  requires a % c == 0, b % c == 0,
  ensures (b - a) % c == 0 {}
```

Second, for problems outside decidable fragments, we enable the SMT solver's nonlinear heuristics in a controlled environment where they are more likely to succeed consistently. Unlike normal Verus assertions, non-linear assertions generate an isolated query without implicit context. In the example below, even though we know from context that `q > 2`, this fact is not automatically available in the assert and must be provided as the premise of the implication. The extra developer burden provides greater predictability.

```
fn f(q: u64, a: u64) requires q > 2 {
 assert(q > 2 ==> (a * a + 1) * q >= (a * a + 1) * 2)
    by (nonlinear_arith);
}
```

Bit-vector Reasoning Low-level bit manipulation is common in systems code, whether to optimize a logarithm estimate via the *count-leading-zeroes* intrinsic or to manipulate flags or bitmaps. Unfortunately, even tools that employ automated solvers often struggle with such reasoning in the context of complex system definitions. In particular, when the solver is asked to reason simultaneously about integers for specification purposes and bit vectors for optimizations, it often fails or, worse, produces fragile proofs that succeed or fail based on minor code perturbations [42].

Hence, multiple system verification projects have developed elaborate encoding styles and proof libraries to manually guide the theorem prover [14, 16, 40]. This eliminates instability, but it leads to a significant developer burden, e.g., requiring tens of proof lines for a simple bit mask [52].

Verus instead carefully separates integer and bit-vector reasoning. Like most tools, Verus maps Rust's integer types (e.g., `u64`) to SMT integers for interoperation with mathematical specifications; it requires the developer to prove the absence of overflow to ensure this is sound.

However, Verus also provides an annotation-invoked mode that automatically interprets Rust's integers as SMT bit vectors when generating an SMT query, eliminating the manual work required by prior projects [14, 16, 40]. Hence, a Verus developer proves bit-manipulation properties by writing, e.g.,

```
assert(x & 511 == x % 512) by (bit_vector);
```

Inside the assertion, `x` is a bit vector, while outside it is treated as an integer, allowing it to integrate with mathematical specifications and reliable SMT-based automation for linear arithmetic. The result is that for the cost of a small annotation, the Verus developer benefits from stable, automated proofs.

Proof by Computation Some proof obligations have obvious, statically computable answers. For example, in a previous project, we tried to verify an efficient implementation of the CRC-32 checksum that used a hard-coded lookup table of precomputed data resulting from complicated polynomial-division operations. Proving that the table resulted from this computation required an excruciating number of proof annotations to guide the solver. We eventually gave up and changed our approach to avoid the hard-coded table.

In Verus, a developer can ask that a proof be discharged by computation [53, 54]: A built-in symbolic interpreter simplifies the expression and sends any remainder to SMT.

Macro-based User-Defined Extensibility Any non-trivial systems project eventually surfaces a repetitive pattern of tedious proof; e.g., proofs that different datatypes can be unambiguously marshalled into byte arrays (§4.2.1). Just as Rust developers exploit macros to reduce repetitive code, Verus developers can do the same to reduce repetitive proofs.

3.4 Automated Reasoning for Multi-Threading

Motivation. No framework for systems verification could be complete without a story for multi-threading. Shared-memory concurrency makes verification significantly more complex, requiring reasoning about all possible thread interactions. Rust's memory safety guarantees extend to multi-threaded code, as captured by its goal of "fearless concurrency." However, it remains far from obvious how to cleanly prove full correctness of concurrent systems.

Our Approach. In Verus, we combine two big ideas.

Idea I: Resource Algebras The first idea is something called a *resource algebra* [55], a proven concept from concurrent separation logic [56]. A resource algebra is a set of rules for creating *ghost resources*, which can help maintain invariants between objects owned by different threads.

Consider a simplified scenario with two objects and the invariant that they agree on the value each stores. Without verification, a developer could informally enforce this invariant by writing code that follows a *protocol* in which an update can only be performed when both objects are obtained by the same thread (as formalized in Figure 4 (top)). The protocol intuitively enforces the invariant, but how can we prove it? During times when the objects are owned by disjoint threads, "who" maintains the invariant? The resource algebra solves this problem, giving us a way to determine sound "update" operations for given invariants.

Rust's ownership types are a natural representation for ownership of resource algebra resources. Indeed, IronSync [29] shows how to do it with Linear Dafny [18]. However, the monoid-based mathematical formalism behind resource algebras is quite technical; it takes considerable expertise to interpret monoids in the context of a concurrent system.

Idea II: A Specification Language for Transitions We make it easier to develop complex resource algebras at scale via a novel framework, VerusSync. We observe that resource-algebra updates are fundamentally *transitions*, the ability to exchange one set of resources for another. In the canonical resource algebra formulation, a developer specifies a monoid and then *derives* transitions by a set of rules.

Our second big idea enables the developer to instead specify transitions directly. VerusSync's syntax is based on the elegant state-transition syntax in Ivy [7, 8], which has been widely used to verify concurrent distributed systems, a close relative of concurrent multi-threaded systems. As with Ivy, VerusSync transitions are described by enabling conditions (specifying when a transition is allowed) and state updates (see Figure 4, bottom). The unique part of VerusSync is its special "sharded" update commands, illustrated below.

```
// --------------- Agreement Protocol ---------------
type Agree;

fn init(val: int) -> (pair: (Agree, Agree))
 ensures pair.0.id() == pair.1.id(),
   pair.0.value() == val && pair.1.value() == val;

fn update(a:&mut Agree, b:&mut Agree, new_value:int)
 requires old(a).id() == old(b).id(),
 ensures a.id() == b.id(),
   a.value() == val && b.value() == val;

fn agreement(a: &Agree, b: &Agree)
 requires a.id() == b.id(),
 ensures a.value() == b.value();

// ------------------- VerusSync --------------------
fields {
  #[sharding(variable)] pub a: int,
  #[sharding(variable)] pub b: int,
}

init!{ initialize() {
  init a = 0; init b = 0;
}}

transition!{ update(val: int) {
  update a = val;
  update b = val;
}}

property!{ agreement() { assert pre.a == pre.b; } }

#[invariant]
pub spec fn agreement_invariant(&self) -> bool {
 self.a == self.b
}
```

Figure 4. Ghost Agreement. *Given the VerusSync code, Verus automatically generates relevant proof obligations, such as the fact that `agreement_invariant` is inductive; in this case Verus's SMT solver dispatches them without additional proof work. Once these proofs succeed, Verus generates the relevant resource, the update operation (`update`), and the proof result (`agreement`). The result is the agreement protocol interface (top part).*

VerusSync is a sound high-level abstraction of monoid-based resources. Our experience has shown us that practical resources are easier to specify by defining transitions than via monoidal composition. Further, in VerusSync, the key proof obligations are *safety conditions*, proven by supplying an inductive invariant. This matches the developer's intuitions about loop invariants and distributed system invariants.

VerusSync In Action. In VerusSync, the developer first constructs a state comprised of fields, each labeled with a *sharding strategy* defining how the field relates to ghost *shards* manipulated in the code. For example, a **variable** field is represented by a single shard; a **map** field is represented by one shard for every key-value entry in the map.[2]

Next, the developer defines the protocol as VerusSync transitions. Each "update" operation has one meaning for the aggregate state and a corresponding meaning as an operation on shards. For example, the **add** keyword adds a key-value pair to a state's **map** field, and correspondingly creates a shard containing the pair. Similarly, **remove** means "remove a key-value pair" and "consume a shard".

[2] The sharding strategies together define the monoid in the underlying resource algebra, a formality VerusSync hides from developers.

NR Queue. At the core of our NR case study (§4.2.2) is a ring buffer that tracks three pieces of state in shared memory:

- a buffer for storing message entries,
- a *tail pointer* where the next message goes, and
- a per-thread *head pointer*, indicating where each thread should read the next message.

There is a complex, multi-step protocol for reading and processing messages from the queue. This protocol is implemented by a thread called the *executor thread*. As the queue operates, no one thread ever simultaneously owns all the pieces of state above, as they are accessed and updated concurrently. Even so, we maintain complex invariants, e.g., relating an executor thread's internal state to its head pointers, or relating the head and tail pointers to non-empty buffer entries.

We model this protocol as a VerusSync ghost resource with fields representing the pieces of state, and we distribute the resulting shards across the threads. The ring buffer's fields are sharded with different strategies.

- The `tail` field marks the next empty space. Its value is represented by a single shard that a thread must own to read or modify the field. The ghost shard is associated with a particular physical memory word accessed atomically, here via compare-and-swap.
- The `buffer_size` field is `constant`: all threads agree on (can "read") its value; it is permanently read-shared.
- The `local_versions` field contains per-thread head-pointers into the queue. It is represented by a map where each entry is an ownable shard, each associated with a different atomically-accessed memory word.
- The `executor` field describes the intermediate state of each executor thread within its multi-step protocol.

In NR, an executor thread pops operations off the queue and processes them. The thread selects a range to read, reads each buffer entry in that range, and finally updates the atomic field corresponding to an entry of `last_version` to point to the end of this range. Figure 5 shows this final step.

The VerusSync approach lets Verus determine, *syntactically*, that the transition only affects two shards. This then allows the developer to perform the operation when they have ownership of those two shards *without* ownership of any of the many other shards in the system. Finally, because VerusSync allows us to easily interpret this system as a transition system, we can use traditional state-machine techniques to prove global properties of the ring buffer, most notably that the threads' accesses to the buffer entries are well-formed.

4 Evaluation

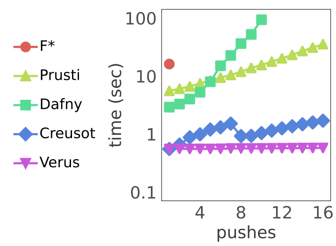
We evaluate Verus as a system verification language across two key dimensions. First, is its proof automation sufficiently powerful and fast for complex system verification? Second, is it expressive enough to specify important system properties and give developers the freedom to write high-performance code that satisfies them?

```
fields {
 #[sharding(variable)] pub tail: LogIdx,
 #[sharding(constant)] pub buffer_size: LogIdx,
 #[sharding(map)]
 pub local_versions: Map<NodeId, LogIdx>,
 #[sharding(map)]
 pub executor: Map<NodeId, ExecutorState>,
  ...
}

reader_finish(node_id: NodeId) {
 // Only finish when executor reaches the end
 require(cur == end);

 // Advance node's state from `Reading` to `Idle`.
 // The `let` indicates a pattern-binding.
 remove executor -= [ node_id =>
  let ExecutorState::Reading(Range{start, end, cur})
 ];
 add executor += [ node_id => ExecutorState::Idle ];

 // Advance the node's current version to `end`
 remove local_versions -= [ node_id => let _ ];
 add  local_versions += [ node_id => end ];
}
```

Figure 5. A More Advanced VerusSync Example. *An executor thread completing an update takes this* `finish` *transition. The transition is only possible if the thread owns two shards needed for* `remove`*: one showing the executor is working on* `Reading` *a range that finishes at* `end`*, and another that proves it owns the right to mutate the* `node_id` *version entry. The transition replaces those shards (via* `add`*) with one that notes the executor is now* `Idle` *and another that stores* `end` *into the* `node_id` *version entry.*

	Single	Double
Verus	0.66	1.15
Creusot	1.88*	30.83
Dafny	3.83	28.11
Low*	7.16	70.17
Prusti	18.80	n/a

(a) Verification time (sec) for the singly linked list and doubly linked list millibenchmarks.



(b) Verification time when varying the number of pushes to four singly linked lists. Note the log-scale y-axis.

Figure 6. Millibenchmarks. *Median of 20 single-threaded runs.* * *This measures the non-interactive time for Creusot to produce a partial result; completing the proof requires manual intervention.*

An ideal evaluation would build K large systems in N different frameworks; however, each such system might warrant an entire paper. Instead, we adopt a pragmatic, multi-scale evaluation strategy. First, we conduct "millibenchmarks" that compare Verus against many verification frameworks on small but representative system-relevant tasks. Second, macrobenchmarks show that these benefits translate to benefits at scale. In total, we verify over 6.1K source lines of Rust code, entailing over 31K source lines of proof. Except where noted, the experiments below are conducted on an AWS EC2 m5.8xlarge instance (Intel Xeon Platinum 8175M CPU @ 2.50GHz) with 16 cores (SMT disabled) and 128GiB of memory.

4.1 Millibenchmark Evaluation

We begin by designing a series of "millibenchmarks", each large enough to capture an important system verification task, but small enough that we can implement them in multiple verification frameworks and analyze them in some detail. We measure the wall-clock time to verify each version. To reduce the risk that we use another verification framework naïvely, we draw our benchmarks from those provided by the frameworks themselves, so we presume they are reasonably optimized. Where we change examples or port new ones to existing frameworks, we have confirmed with the framework's designers that our artifacts are reasonably idiomatic.

4.1.1 Frameworks We focus on comparison to verification frameworks that **(a)** have been used to verify complex properties of large-scale systems, and **(b)** offer a large degree of automation "out of the box". This includes **Dafny** [34], which has verified cryptographic code [51, 57], application stacks [29], distributed systems [15], storage systems [18], and production-scale multi-threaded systems [29]; and **F*** [35] (specifically the Low* subset [58]), which has verified ~43K lines of C and assembly code in a cryptographic provider [16], the TLS 1.3 [59] and QUIC [60] record layers, and the Signal messaging protocol [61]. We also include **Ivy** [7, 8], which has verified distributed system protocols [9, 20–24], as a representative of tools trading expressivity for proof automation.

To confirm that Verus's benefits arise from design decisions beyond simply building on Rust, we include two state-of-the-art automated Rust verification frameworks, **Prusti** [62] and **Creusot** [63], even though they do not verify concurrent code and have not yet been applied to large-scale systems projects.

4.1.2 Millibenchmarks We first define three general-purpose millibenchmarks on basic data structures. Even these simple examples fall outside the restricted logics of Ivy [8] or Serval [5], so we add a benchmark that fits in Ivy's EPR.

Singly linked list. To evaluate verification efficiency on a small task, we verify that a singly linked list implements an abstract sequence. The verified API is consistent across the verification tools. The list supports pushing at the head, popping at the tail, indexing, and iteration.

Doubly linked list. To evaluate how verification performance scales with complexity, we implement a doubly linked list and prove it implements an abstract sequence, which requires `unsafe` Rust because of its cyclic pointers. This list supports pushing and popping at both ends and iteration.

Memory reasoning. Reasoning about memory updates at scale is a perennial challenge for system verification. Hence, using the verified lists, we evaluate the cost of memory reasoning by repeatedly updating four instances of the list within the same function. This requires the verifiers to determine whether an update to one list might affect another.

Distributed lock. To evaluate distributed protocols, we port a distributed lock to Verus and prove mutual exclusion in two ways: in default-mode following the Dafny proof [15], and using Verus's EPR mode, similar to the Ivy proof [7].

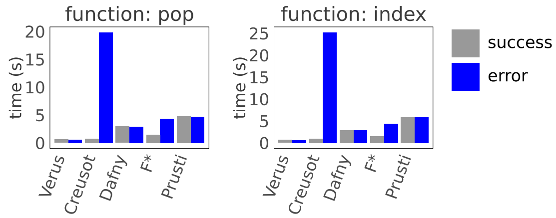


Figure 7. Error Results. *Median of 20 runs. Tools use 8 threads.*

4.1.3 Millibenchmark Results

Linked lists. Figure 6a shows the verification time for our two linked list examples. For the singly linked list, we see that the other frameworks take 3–28× longer than Verus, while for the more complex doubly linked list, they take 24–61× longer. A major cause is Verus's emphasis on generating concise SMT queries (§3.1, Figure 8). Prusti cannot express cyclic pointers.

Successful verification times are the easiest apples-to-apples comparison, but in reality, developers most often wait on tools for failure feedback. To capture this, we "break" each singly-linked-list proof twice, once by removing a precondition in `pop` and once in `index`, and measure the time for the tool to report an error. Figure 7 shows that Verus, Dafny, and Prusti pinpoint failures as quickly as they report success. F⋆ degenerates from one second to four. Creusot's approach degenerates from one second to twenty.

Memory reasoning. Figure 6b shows how the verification frameworks handle increasing numbers of memory modifications to four singly linked lists. Dafny and F⋆ must perform complex aliasing reasoning, and hence Dafny's verification time grows dramatically as memory modifications increase. F⋆ struggles even more, failing to return after more than one push. Prior Rust-based tools perform better, but they still grow super-linearly,[3] whereas Verus remains linear (with a slope of ~ 1.6ms/push) across the entire benchmark. The results for doubly linked lists are similar, with Verus remaining linear with a slope of ~ 1.8ms/push.

Distributed lock. The safety proof uses an inductive invariant maintained across protocol steps. The default-mode proof of inductiveness is around ~25 lines. When abstracted into EPR, the proof is automatic, but surprisingly, creating and using the abstraction required ~100 lines of (straightforward) boilerplate. This suggests that **(a)** EPR benefits complex examples, like the Delegation Map in §3.2, more than simple ones, and **(b)** Verus needs to automate the boilerplate.

4.2 Macrobenchmarks

To show that the benefits of Verus identified by the millibenchmarks translate into benefits at scale, we assembled a suite of macrobenchmarks. First, to compare with other frameworks at scale, we port two large verified systems to Verus (§4.2.1,§4.2.2) and compare each to its original.

[3]Creusot "races" several solvers in parallel; the dip in the graph shows when a different solver starts to "win".

System → Verifier	Line Count: trusted	proof	code	P/C Ratio	Time (s): 1 core	8 cores	SMT (MB)
IronKV							
→ Verus	1613	4509	1533	2.9	41	18	17
→ Dafny	1205	8070	1923	4.2	445	201	352
NR							
→ Verus	369	5237	736	7.1	17	9	22
→ L.Dafny	104	7828	730	10.7	1089	228	2063
Page table	1145	5304	394	13.5	59	28	49
Mimalloc	282	13703	3178	4.3	262	55	152
P. log	1284	2913	739	3.9	12	6	9
Verus total	4692	31231	6119	5.1			

Figure 8. Macrobenchmark Statistics. *Verification performance and proof overhead of each benchmark, including the original verified systems we ported to Verus. Times are seconds to successfully verify the entire project. SMT is the total size of queries sent to Z3.*

Second, since developers rarely start from completed code and proofs, we examine the tool's behavior in its primary mode, when verification *fails*. To explore fresh development, we report on the experience of writing and verifying three additional systems from scratch in Verus (§4.2.3-§4.2.5).

4.2.1 Porting IronKV from IronFleet [15]

IronFleet [15], originally developed in Dafny, allows developers to prove that a distributed system's implementation is both safe and live. This requires proofs about both the implementation that runs on each host, and the protocol the hosts use to achieve the system's high-level properties.

Verification Target. We port the host implementation of IronFleet's IronKV, which dynamically shards a key-value store across a set of nodes. We skip the protocol level, since Dafny and Verus share similar mathematical modeling tools and are likely to admit very similar proofs. We translate the protocol-level host description to Verus as the goal spec, implement the host in Rust, and prove it matches this spec.

Porting Experience. To support comparisons, our port preserves IronKV's algorithmic decisions, but, where highlighted below, our design exploits Verus-enabled improvements.

<u>*Basic Improvements*</u> We encountered multiple places where IronFleet split a simple task across many functions, presumably to keep verification times manageable. For example, IronFleet's `MaybeAckPacketImpl` accepts a message and looks up its sequence number in a tombstone table to decide whether to ack it. In IronFleet, this simple task is split into three functions across 37 lines. Our port inlined those three functions into a single 30-line Verus function that verifies faster than any of the three original Dafny functions.

In another place, the IronFleet code dealt with the painfulness of reasoning about fine-grained mutation by replacing an entire data structure with a modified version, leading to inefficient (and confusing) code. The Verus port simply uses an

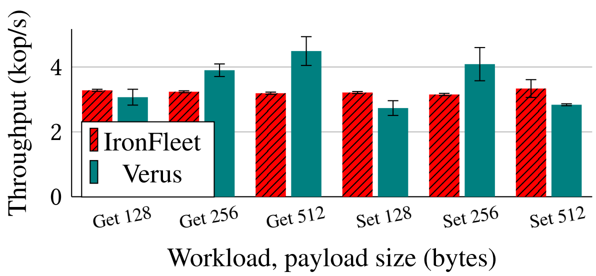


Figure 9. IronKV Performance. *The Verus version performs comparably to the IronFleet original. Each bar shows the mean of 100 trials; error bars show 95% confidence intervals.*

`&mut` reference, which expresses the original intent directly and avoids a performance penalty.

<u>*Parsing and Marshalling*</u> IronFleet's IronKV includes a generic marshalling library for basic types: arrays, tuples, tagged unions, 64-bit integers, and byte-arrays. The developers mapped Dafny datatypes to and from these basic types, manually constructing tedious boilerplate code and proofs.

We instead wrote our own marshalling library that eliminates this tedium via user-defined macros (§3.3) and provides a more ergonomic interface using traits. Our `Marshallable` trait (including a marshaller, parser, and relevant lemmas), provides a consistent interface, improving on the original which relies on naming convention. Primitives (like `u64`) and repetition (`Vec<T>`) implement this trait with hand-written proofs. Arbitrary `structs` and `enums` use macros that automatically derive implementation and proofs, eliminating the repetitive manual proofs prevalent in the Dafny original.

<u>*EPR Delegation Map Proof*</u> Using Verus's EPR mode drastically simplified the proof of this data structure (§3.2).

Evaluation. Figure 8 shows that the Verus port saves both code and proof, improving the proof-to-code ratio. The *trusted* column is noisy because of how we reinterpreted the original protocol layer as our spec. The authors that completed this port reported crisp interactivity, supported by the 95% smaller query sizes and 10× faster verification.

To confirm the fidelity of our port, we benchmark both systems with the test harness from IronFleet's repository [64]. The experiments run on Windows 11 Enterprise on a 2.4GHz Intel Core i9-10885H CPU 8-core laptop with 64GiB of RAM. We launch three server processes on separate ports, then launch the client workload generator with 10 threads and 10,000 keys for 30 seconds. We vary the workload (Get vs. Set) and the payload size; sizes are limited to 512 bytes by current IronFleet repository. Since our port is fairly faithful, we anticipated similar performance, which Figure 9 confirms.

Summary. Our success porting the IronKV host demonstrates that Verus at least subsumes the functionality of Dafny employed by IronFleet, and that it is largely compatible with that approach to system verification. It improves by moving heap reasoning into ownership reasoning, which eases performant implementation of mutable data structures and, by optimizing solver performance, enables coalescing tasks into more reasonably-sized functions. Rust's macro system and Verus's EPR support significantly reduce developer tedium.

4.2.2 Porting NR from IronSync [29]

The IronSync framework [29] enables verifying the correctness of complex shared-memory programs that employ application-specific synchronization primitives to achieve high performance. It is built on Linear Dafny [65], a variant of Dafny extended with simple Rust-inspired ownership types.

Verification Target. We ported the IronSync implementation of the Node Replication (NR) concurrency library [66]. NR converts a sequential data structure into a high-performance, concurrent version via replication and flat combining. We prove the same result as IronSync, namely that the concurrent system meets the sequential functional spec *linearizably*.

Porting Experience. Our Verus-NR implementation is more faithful to the original NR than IronSync-NR in three ways. First, both NR and Verus-NR are written in Rust; IronSync-NR was a port to Dafny. Second, Verus-NR exposes a trait-based interface similar to NR's in order to support generic data structures, whereas IronSync does not support traits. Finally, Verus-NR admits runtime-defined counts and dynamic thread registration, whereas IronSync-NR fixes the replica and thread counts statically.

Furthermore, proofs related to concurrency are substantially simplified in Verus-NR due to the use of VerusSync (§3.4) rather than IronSync's monoid formalism. VerusSync allows cleaner, application-level reasoning, and the simplification is reflected in our reduction in proof code.

Evaluation. Figure 8 shows that Verus-NR requires far fewer lines of proof. This is mostly due to the use of VerusSync. Verus also improves the verification time by *two orders of magnitude*. Such speed creates a qualitative advantage: Where we might verify one function at a time with a slower tool, we iterated while verifying the entire project, which provides an early warning when a change to a function contract breaks a proof elsewhere.

We compare Verus-NR's performance with unverified NR and with IronSync-NR using IronSync's artifact [67] benchmark. We run this benchmark on a four-socket Intel Xeon Platinum 8260 with 24 cores and hyper-threading enabled. The benchmark initializes NR with four replicas wrapping an x86-page-table data structure. We increase the thread count, filling up NUMA cores before utilizing hyperthreads. We measure the throughput at write ratios of 0%, 10%, and 100%. Figure 10 shows that Verus-NR's performance matches the unverified, highly optimized original implementation.

Summary. Porting NR shows that Verus can verify state-of-the-art concurrent data structures optimized via application-specific synchronization primitives. It does so faster, more intuitively, and with less developer tedium than existing state-of-the-art concurrent verifiers.

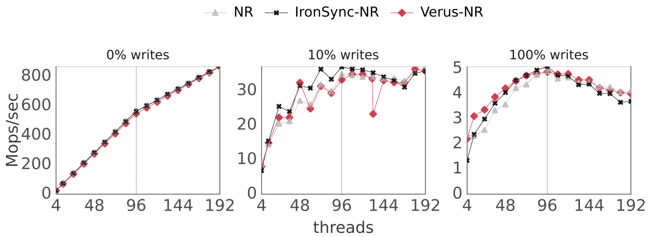


Figure 10. NR. *Verus-NR matches the performance of IronSync-NR (and the unverified original) despite a much easier proof.*

4.2.3 New Verified System: An OS Page Table

To evaluate Verus when developing a new verified system, we implement a verified page-table data structure for x86-64.

Verification Target. The map and unmap operations of a page table entail traversing and modifying a tree data structure whose nodes pack flags and addresses into 64-bit machine words. Verifying it requires specifying and reasoning about ISA-level software and hardware components.

Correctness is specified from the perspective of a user-space process on a single-processor system: reads return the most recently written value; map and unmap operations expand and restrict the virtual memory domain. The implementation employs a (trusted) hardware spec that defines how the MMU interprets page table memory to translate virtual addresses to physical.

Development Experience. We highlight two aspects of how Verus enables necessary low-level reasoning: bit-level manipulation of 64-bit words and specifying how the implementation may interact with page-table memory.

Page-table entries are bit-packed 64-bit words. To reason about them efficiently, we rely heavily on Verus's automation for bit-vector, non-linear, and proof-by-computation reasoning (§3.3), which we invoke 62, 39, and 11 times, respectively. They enable us to automatically discharge conditions like:[4]

```
forall|a:u64,i:u64| i < 13 && (a & mask!(13u64,29u64) ==
   0) ==> ((a | bit!(i)) & mask!(13u64,29u64) == 0)
```

which in other frameworks [34, 35] would have incurred tedious manual proof [16, 52].

The trusted spec of the MMU describes how the it translates memory accesses based on the values of page table entries. This interpretation is only meaningful with respect to the physical values in the memory storing the page table. The trusted spec provides a struct that encapsulates ownership (and allocation) of the page-table memory: ownership prevents other writes to the entries, the encapsulation tracks the values of the entries as ghost state, and the MMU contract can thereby make promises about its translation. Ownership facilitates soundly specifying this hardware behavior.

Evaluation. As shown in Figure 8, our page table consists of 394 lines of executable code, which required 5304 lines of proof, resulting in a relatively high 13.5:1 proof-to-code ratio. This may be a result of this being our first large-scale development in Verus; more experience may suggest different abstractions for some proofs. A page table is also a complex OS component, so a high ratio may be inevitable. When it comes to verifying a complete OS, independent work uses Verus to verify a full microkernel [26]; the authors report a 7.5:1 proof-to-code ratio.

[4]`mask!(x, y)` sets the bits between x and y.

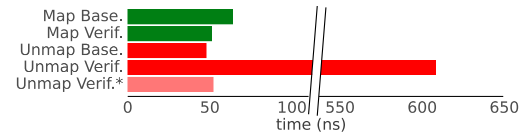


Figure 11. Page Table Run-time Performance.

We compare our implementation against a recent unverified page table implementation [66] in a single-threaded setting, reporting mean latency over 100M map and unmap operations on 4K frames. Figure 11 shows our implementation matches the reference for mapping frames, but our unmap is much slower. We run this benchmark on a Intel Xeon E-2224G with frequency scaling disabled. This discrepancy is because we reclaim emptied page directories, which we confirm by benchmarking an (unverified) modification of our page table with reclamation disabled (Unmap(Verif.*) in the figure). Larger OS-level benchmarks show negligible differences, even with the cost of our unmap.

Summary. This system demonstrates Verus's ability to specify both OS and hardware interfaces and reason about a complex low-level implementation connecting the two.

4.2.4 New Verified System: Memory Allocator

As our largest, most complex new system, we develop a concurrent memory allocator.

Verification Target. Our work is based on *mimalloc* [68], which provides state-of-the-art performance. Since mimalloc is written in C, our Verus version translates C into Rust idioms, but preserves the overall data structures and algorithms. We prove our implementation functionally correct, meaning that every allocation returns non-aliased memory.

Development Experience. Any memory allocator system faces two significant challenges.

Address space management. The overall objective of a memory allocator is to bridge the gap between an OS memory API that supports *coarse-grained*, page-aligned allocations (e.g., the `mmap` syscall on Linux) and an allocator API supporting *arbitrary-sized* allocations (`free` and `malloc`). This requires careful accounting of the address space.

To reason about the address space, we use ghost memory permissions [10]. We write a trusted specification for `mmap` in terms of ghost permissions; these permissions can then be passed up to the client when they call `malloc`.

Cross-thread deallocations. It is possible that the client frees memory on a different thread from which it was originally allocated. Mimalloc's design handles this by depositing cross-thread deallocations into an *atomic free list*, a lock-free linked list accessed via atomic compare-and-swap.

Benchmark	mimalloc	Verus-mimalloc
cfrac	4.6 s.	9.7 s.
larsonN-sized	4.1 s.	12.0 s.
sh6benchN	0.14 s.	2.0 s.
xmalloc-testN	0.34 s.	0.73 s.
cache-scratch1	1.2 s.	1.2 s.
cache-scratchN	0.16 s.	0.16 s.
glibc-simple	1.2 s.	6.6 s.
glibc-thread	1.1 s.	3.6 s.

Figure 12. Mimalloc Benchmarks Supported by Verus-mimalloc. *Benchmarks run on Linux on an 8-core, 3.60GHz Intel i9-9900K. The mimalloc authors label cfrac and larsonN-sized as "real world" benchmarks and the others as pathological stress tests.*

To do the same, we utilize Verus's ability to associate ghost state with atomic locations (§3.4). Specifically, we deposit ghost memory permissions from cross-thread deallocations into the atomic variable holding the linked list's head pointer.

Evaluation. We have implemented a subset of the features and optimizations supported by mimalloc's design. For comparison's sake, Verus-mimalloc has about 3.1K lines of executable code, while the original is about 10K lines of code. Our allocator supports use as a drop-in replacement for the system allocator with some limitations: it does not yet support `realloc`, aligned allocations, or allocations greater than 128KiB. It can complete 8 out of 19 benchmarks from mimalloc's benchmark suite [69], though it does not yet reach performance parity (Figure 12). We focused our initial development on aspects highlighted in the mimalloc report [68], particularly those affecting concurrency, so we believe Verus-mimalloc is prepared to support the missing features and optimizations in the future.

Figure 8 shows a favorable proof-to-code-ratio of 4.3. Furthermore, the allocator's user-facing specification is very succinct: between initialization, `malloc`, and `free`, it is only 37 lines. The allocator relies on OS interfaces (`mmap` and thread utilities) with 245 lines of trusted spec, bringing the total to 282. Our allocator also relies heavily on Verus's bit-vector, non-linear, and proof-by-computation automation (§3.3), which we invoke 78, 71, and 187 times, respectively.

Summary. Totaling over 17.2K lines (code and proof), Verus-mimalloc is the largest of our macrobenchmarks and the largest Verus project we know of. Even so, Verus verifies the entire project in just over a minute.

4.2.5 New Verified System: Persistent Log

To evaluate Verus's utility for verifying production code, we developed a persistent circular log for byte-addressable storage devices such as Optane DC Persistent Memory [70].

Verification Target. The log offers asynchronous `append` and synchronous `advance_head` operations to its storage system client, and supports atomic appends to multiple separate logs.

Our log is designed for storing low-level metadata and data in a cloud-scale production storage system. It is integrated into a production codebase, which incorporates it via `Cargo.toml` as just another Rust crate.

Development Experience. We verify the implementation refines an abstract, infinite log; that all operations are atomic with respect to crashes; and that the log metadata is protected from corruption up to CRC. These properties are essential for persistent memory, which has a small persistence granularity and is at risk for fine-grained media errors, random bit flips, and stray writes [71]. They are also especially valuable in cloud-scale storage, where crashing and corruption bugs too rare to detect with traditional testing still turn up.

Production integration is simple: Verus erases all but Rust content for tools other than the verifier; standard `rustc` sees only executable code and readily consumes it. For the crates that the verified code depends on, such as a CRC crate, we write a specification and mark it trusted.

To simplify our proofs, our initial verified log converted each metadata structure to a byte slice before writing to persistent memory, incurring unnecessary copying in DRAM. Our latest version provides a `Serializable` trait with `spec` methods to specify the byte-level layout of metadata. Structures that implement this trait can be copied directly to persistent memory without runtime conversion to a slice, removing this overhead while providing the same guarantees.

Evaluation. Verus verifies the log implementation in 12s with a proof-to-code ratio of 3.9 (Figure 8), while offering correctness, crash safety, and metadata-corruption detection.

We evaluate performance on a 128GiB Optane Persistent Memory Module device. All log updates are written directly to the device through a 4GiB memory-mapped file in Ext4-DAX [72]. We compare the latest verified log against `libpmemlog` [73] from the state-of-the-art PMDK [74], and against the original log prototype.

Figure 13 compares the append throughput of all three systems with 95% confidence intervals. Each data point is 8GiB of appends (including operations to free space so the log can wrap around). The initial version of the verified log provided low throughput on small appends due to its extra copying; the latest version eliminates this overhead and achieves comparable throughput to `libpmemlog`. `libpmemlog` and the current verified log have similar throughput even though the verified log calculates CRCs and `libpmemlog` does not, because `libpmemlog` acquires and releases a lock on each append while the verified log uses no locks.

Summary. The verified log shows that Verus can develop software that integrates naturally into production code bases. It also demonstrates that Verus supports reasoning about domain-specific properties, like crash safety, without "baking" such reasoning into Verus itself.

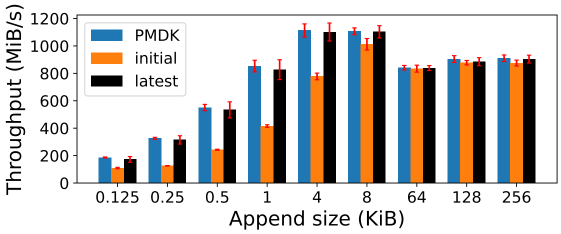


Figure 13. Verified log vs. `libpmemlog`. *Append throughput*

5 Related Work

Multiple groups [2, 3, 75–80] have employed *interactive proof assistants* like Coq [31] and Isabelle/HOL [81] to verify systems. By default, these are less automated than SMT solvers: developers manually walk the verifier through the proof by applying tactics, leading to large proof-to-code ratios (e.g., over 20:1 for the initial version of seL4 [82]), despite some recent domain-specific improvements [80].

In contrast, Verus continues a line of work on *program verifiers* [13, 34, 35, 62], which focus on verifying programs written in a particular language. Using solvers, these tools typically offer more automation "out of the box" (e.g., the Ironclad project [14] had a proof-to-code ratio of 5:1).

Verus benefits from prior work on using ownership type systems to simplify memory reasoning in systems code. For example, a prior study on Linear Dafny [18] quantifies the benefits of using ownership to reason about memory access in complex systems [65], and shows how ownership can co-exist with traditional memory reasoning. Linear Dafny's ownership types are considerably less sophisticated than Rust's, however. Linear Dafny in turn builds on Cogent's purely functional support for borrowing [83] and on early work by Wadler [84].

Verus is one of several tools building on Rust. RustBelt [85] focuses on manually verifying unsafe Rust code using Coq. Prusti [62] encodes Rust code into the Viper verification framework [86], which leads Prusti to essentially re-verify Rust's typechecking, resulting in larger SMT queries. While conceptually similar to Verus, Creusot [63] lacks Verus's ability to reason about ghost resources. As §4.1 shows, Verus verifies equivalent code faster than either Prusti or Creusot. Aeneas [87] translates Rust code into a pure functional form that the developer then reasons about in a separate proof assistant (currently Lean [32]); this differs from Verus's intrinsic approach where the developer writes code and proofs directly in Rust. In addition, Prusti, Creusot, and Aeneas do not offer ERP-style automation, include support for system-specific idioms, or reason about concurrency.

6 Conclusion

Verus aims to consolidate the gains made by the system-verification community in a unified tool. By building on a mainstream language (Rust), Verus makes these gains available to a much broader audience of system developers. At the same time, by leveraging Rust and our carefully designed system-oriented features, Verus provides a higher-level starting point for future research in this area. Ultimately, we hope that Verus enables researchers to identify the exciting research challenges that emerge as we scale verification to new heights of system size and complexity.

References

[1] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[2] 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 ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, California, October 2015.

[3] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

[4] 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 Symposium on Operating Systems Principles*, 2017.

[5] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[6] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[7] Oded Padon, Kenneth L. McMillan, Mooly Sagiv, Aurojit Panda, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[8] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2020.

[9] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[10] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 2023.

[11] Nicholas D. Matsakis and Felix S. Klock. The Rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[12] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[13] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

[14] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2014.

[15] 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 ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[16] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cedric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[17] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACL×N: Verified generic SIMD crypto (for all your favorite platforms). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2020.

[18] Travis Hance, Andrea Lattuada, C. Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[19] R. Piskac, L. M. de Moura, and N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4), 2010.

[20] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2017.

[21] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In Tim Brecht and Carey Williamson, editors, *Proceedings of the ACM Symposium on Operating Systems Principles, (SOSP)*, 2019.

[22] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2021.

[23] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2022.

[24] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[25] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *To Appear in Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2024.

[26] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere: Towards practical verified kernels in Rust. In *Proceedings of the Workshop on Kernel Isolation, Safety and Verification (KISV)*, 2023.

[27] Ziqiao Zhou, Weiteng Chen, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *To Appear in Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2024.

[28] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in Rust. https://arxiv.org/abs/2311.03739, 2023.

[29] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2023.

[30] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2023.

[31] Coq Development Team. The Coq Proof Assistant https://coq.inria.fr/.

[32] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *Proceedings of the Conference on Automated Deduction (CADE)*, August 2015.

[33] 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*, 2002.

[34] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[35] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages*, 2016.

[36] Liam Proven. Linux 6.1: Rust to hit mainline kernel. https://www.theregister.com/2022/10/05/rust_kernel_pull_request_pulled/, October 2022.

[37] Shane Miller and Carl Lerche. Sustainability with Rust. https://aws.amazon.com/blogs/opensource/sustainability-with-rust/, February 2022.

[38] Google. Announcing KataOS and Sparrow. https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html, October 2022.

[39] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. RUDRA: Finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[40] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Automated verification of concurrent code with sound semantic extensibility. *ACM Transactions on Programming Languages and Systems*, 44(2), June 2022.

[41] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[42] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*, October 2023.

[43] Leslie Lamport. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[44] Robert Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, 1967.

[45] Tony Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.

[46] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[47] Michał Moskal. Programming with triggers. In *Proceedings of the Workshop on Satisfiability Modulo Theories*, 2009.

[48] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *PACMPL*, 2(POPL):26:1–26:33, 2018.

[49] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 13243, 2022.

[50] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[51] Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. Galápagos: Developing verified low-level cryptography on heterogeneous hardwares. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2023.

[52] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada code repository. https://github.com/microsoft/Armada/blob/ee799110f9aecc3deab31b94521bdbcd27f58363/Test/qbss/bv.dfy#L54, December 2023.

[53] Martijn Oostdijk and Herman Geuvers. Proof by computation in the Coq system. *Theoretical Computer Science*, 272(1–2), February 2002.

[54] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, April 2019.

[55] Ralf Jung, R. Krebbers, Jacques-Henri Jourdan, A. Bizjak, L. Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[56] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3), April 2007.

[57] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, August 2017.

[58] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in $F^{\star}$. *PACMPL*, 1(ICFP), September 2017.

[59] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jiangyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. Implementing and proving the TLS 1.3 record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[60] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021.

[61] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019.

[62] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), October 2021.

[63] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verication of Rust programs. In *Proceedings of the International Conference on Formal Engineering Methods*, October 2022.

[64] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Haojun Ma, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet code. https://research.microsoft.com/projects/ironclad/,

2015.

[65] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 2022.

[66] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[67] Travis Hance and Yi Zhou and Andrea Lattuada and Reto Achermann and Alex Conway and Ryan Stutsman and Gerd Zellweger and Chris Hawblitzel and Jon Howell and Bryan Parno. IronSync OSDI 2023 artifact. https://github.com/secure-foundations/ironsync-osdi2023, 2023.

[68] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[69] Daan Leijen. Mimalloc-bench. https://github.com/daanx/mimalloc-bench, 2023.

[70] Intel. Intel optane persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[71] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the ACM Symposium on Operating Systems Principles, (SOSP)*, 2017.

[72] Linux Kernel Developers. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[73] PMDK Developers. libpmemlog. https://pmem.io/pmdk/manpages/linux/v1.3/libpmemlog.3/.

[74] PMDK Developers. PMDK. https://pmem.io/pmdk/.

[75] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2014.

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

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

[78] 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 ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2015.

[79] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2016.

[80] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling machine-checkable systems verification in Coq. In *Proceddings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2023.

[81] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[82] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[83] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[84] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

[85] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL), January 2018.

[86] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2016.

[87] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022.