Applying Modern Verification Techniques to a Root-of-Trust Bootloader

Nicholas Gordon
nicholas.gordon@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Carsten Weinhold
carsten.weinhold@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Abstract

Verification tools have become more approachable over the last few years, especially those that allow developers to write proof annotations in the target programming language they already know. We applied one such tool, Verus, to the root-of-trust bootloader of an embedded platform. We report from the point of view of a systems developer, who wants to use verification as a quality-assurance tool for an existing codebase, which was not written with verification in mind. We discuss how we specified a key security property using pre- and post-conditions in the source code, guiding Verus to prove that this property does indeed hold.

Our experiment shows that existing systems code can be retrofitted to enable correctness proofs with Verus, even without prior experience in software verification. Proof annotations increased the bootloader's code size by slightly more than 50 percent. This increase is significant, but did not result in a corresponding increase of binary size or noticeable performance degradation. However, as we applied Verus to this low-level code base, we also encountered several difficulties that required workarounds. We discuss these issues and suggest quality-of-life improvements that could make the use of Verus easier in the future.

ACM Reference Format:
Nicholas Gordon and Carsten Weinhold. 2025. Applying Modern Verification Techniques to a Root-of-Trust Bootloader. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3764860.3768324

1 Introduction

Recently, program verification has matured substantially in the area of systems software, particularly using the Rust language for its safety properties, which complement well the aims of formal verification. Previously, formal verification was considered a useful but prohibitively expensive assurance technique, requiring enormous developer effort or requiring the use of complex proof assistants with unfamiliar languages.

Verification has in the past required too much effort to be considered practical for systems software [16], requiring "heroic" [17] amounts of developer effort. With modern techniques, we believe the cost-benefit ratio puts it within reach of ambitious systems programmers for their own programs, mirroring in a way the transition from languages like C/C++ to safer ones like Rust. Formal verification allows the programmer to reason about abstract *specifications* instead of concrete code, which is especially useful in trusted computing contexts, where security is often based on careful reasoning about a system's properties.

One of these systems, Verus [17], uses proof code that is written in an extended Rust syntax, enabling the co-mingling of program code and proof code. This appears to greatly lower the barrier to entry for systems programmers, and works using Verus claim improved proof-to-code ratios and accessibility as advantages, in addition to the conventional verification claims [7, 8, 17]. However, most works start from scratch when implementing verified code, focusing on writing "proof-oriented code." In the hands of a skilled team, this approach seems to optimize the proof-to-code ratio and "elegance" of the source, but doesn't explore the viability of applying Verus to existing software that was written *without* verification in mind. A successful application of Verus would further the idea that we are nearing a "tipping point" for systems software, pointing the way for more trustworthy and reliable systems.

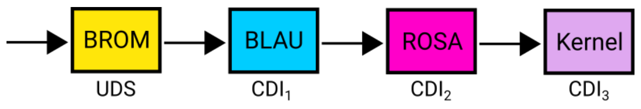

Figure 1. The staging of a root-of-trust bootloader. Starting with a unique device secret (UDS), each stage measures the next one and adds this secret to the bundle (CDI_{n+1}) for the next stage. Each stage deletes its own secrets before switching to the next.

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768324

To explore this, we verify a key property of a root-of-trust bootloader using Verus, exploring its utility for existing codebases in Rust. We specifically select a bootloader to see the applicability to low-level software. Bootloaders are simple, direct pieces of software with less abstraction than even a kernel, often lacking memory allocators, address space isolation, or other abstractions; a kernel *bridges* low and high abstraction, a bootloader *lives* at low abstraction. Other works have investigated other contexts, including higher-level applications [7], simple microkernels [8], and the individual components of larger kernels [6, 17].

Our bootloader, shown in Figure 1, implements the DICE protocol [1] which allows low-powered devices without specialized hardware such as TPMs to establish a layer-by-layer cryptographic identity which is useful for remote attestation purposes. Key to the integrity of this protocol is the *secure erasure* of certain secrets between boot stages. Verifying that our implementation correctly implements this part of the protocol is an important first step toward verifying the whole bootloader. In the end, we show that our bootloader correctly copies the context to the next stage and instructs a trusted assembly routine to do final stack clearing and control handoff. We show how applying modern verification techniques increases confidence in *existing* systems software without requiring costly total verification.

We make the following contributions:

1. We develop a simple specification of context erasure for the DICE protocol in Verus and partially verify a root-of-trust bootloader.
2. We show the feasibility of verifying existing "low-abstraction" code in a modern verification framework, in the context of Rust.
3. We quantify the cost to retrofit existing software with Verus in terms of both its proof-to-source ratio, but also the ratio of pre-verification source to post-verification source, representing the raw "overhead" verification with Verus introduces.
4. We explore the capabilities of Verus and identify deficiencies.

2 Background

Formal verification relieves the programmer of reasoning directly about code and allows reasoning about the *specification* of the code instead. Developing a specification is itself a significant task, but is easier to reason about than the concrete code it specifies, as code deals with the realities of hardware and language semantics. In trusted computing this distinction is of increased importance, since careful reasoning is required to establish that a system is secure through specific properties. These properties are specifications, not code, for example "some secret s is not exposed to an adversary."

Verified systems software is a developed field, including landmark works of "heroic" developer effort to verify whole systems [16]. Numerous Rust-based verification frameworks have been developed. Among them is Verus and although new and under continuous development, has already been used in several projects [7–10, 18, 20, 24, 25].

2.1 Rust-based Frameworks

Owing to its expressive linear type system and borrow-checker, Rust is used in several verification frameworks which span the spectrum in terms of capability, their supported subset of Rust, and proof automatedness. RustBelt [15], Aeneas [14], and RefinedRust [12] focus on providing so-called *foundational* proofs. These foundational proofs are automated with "tactics" and may require more programmer effort, but are machine-checked for correctness. On the other hand, Prusti [4], Creusot [11], Kani [23], and Verus [17] use SMT-based solvers to automate checking and verification. SMT solvers are programs which can solve certain kinds of logical statements, but not all kinds. In particular, they are less expressive than higher-order logic provers like Rocq[1] [22] or Isabelle [19]. In general the trade-off between foundational and SMT-based provers is of expressiveness versus proof accessibility, as the approach to proving statements using a framework like Verus is more similar to programming than developing a proof in a solver like Rocq. We selected Verus for its combination of: accessible syntax, well-developed language support, and acceptable expressivity. All of these frameworks vary in terms of expressivity, such as whether they cover unsafe Rust code or how detailed the memory model is, and the subset of the Rust language constructs they support.

2.2 Verified Systems Software

Several milestone works exist which verify systems software. Among the two largest-scale are the seL4 microkernel, which paired C with Isabelle [16] and the CertiKOS kernel/hypervisor, which paired C with Rocq [13]. Both projects claim sizes less than 10k lines of source code, but both have proof sizes beyond 100k lines of proof code. This high 10:1 proof-to-source ratio characterizes the current state of the art in verifying systems software, though improving this ratio is a main claim of Verus.

Several works have verified systems software using Verus, a selection of which is compared in Table 1: VeriSMo, a security module for AMD-SNP [25], the Atmosphere microkernel [8], the Vest serialization framework [7], and a handful of other OS components [17], including a page table manager and memory allocator. These works have proof-to-code ratios from 2.5 up to 13.5, showing significant improvement in proof burden at the best end and comparable to existing strategies at the worst end. From these we infer that while

[1]Previously known as Coq

Project	*Proof-to-Code Ratio*
VeriSMo [25]	"approximately" 2:1[2]
Vest [7]	~2:1
Memory Allocator [17]	4.3:1
Atmosphere [8]	7.5:1
Page Table Manager [17]	13.3:1

Table 1. Proof-to-code ratios of Verus projects sometimes improve over existing frameworks and deliver very low proof-to-code ratios.

Verus *enables* lower proof-to-code ratios, it of course does not *guarantee* them, and that further work should explore whether one of Verus's central advantages applies to other contexts.

3 Approach

We define our goal state, create a first specification, and note our assumptions and limitations by explaining what code remains trusted, for reasons both at the project level and the broader Verus level.

3.1 Verification Goal

Plainly said, we aim to prove that each program in our bootloader chain deletes its own session data. We assume, as Verus does, that the compiler is correct and not malicious. We assume that the secrets are *not* copied elsewhere, e.g. by side-stepping Rust's ownership type semantics. To be clear, we are using Rust's standard ownership semantics, which ensure that there is only one owner of a value at a given time. Additionally, types in Rust are not copyable unless they are marked with `Copy` or `Clone`; we do not implement these traits for our secrets. However, using unsafe code it is possible to bypass these ownership semantics and directly access the underlying value. We assume that each program correctly *derives* those context secrets. As availability is less important than confidentiality for our root-of-trust model, we leave verification of these additional properties as future work.

3.2 Specification

Fully and correctly specifying a program is already a substantial amount of work and a well-researched topic. To focus on the interaction with system software, we develop a narrow specification aimed at capturing a key property for our root-of-trust bootloader. Our bootloader implements the DICE protocol [1], which has already been proven formally correct using F* in another work [21].

Figure 2 visualizes what each program ("layer") in the boot-chain does:

[2]Only approximate numbers were given in their text.

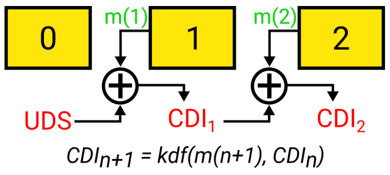

Figure 2. Each stage$_n$'s secret is generated by the stage$_{n-1}$ before it with a secret$_{n-1}$ and a measurement$_n$. Before switching layers from n to $n+1$, CDI_n must be securely deleted.

1. Uses the secret bundle (CDI_n) from the previous layer to derive a key k_n.
2. Loads and measures ($m(n+1)$) the next layer, and uses this measurement plus its key k_n to generate the secret bundle for the next layer (CDI_{n+1}).
3. For layers after the first one, sign CDI_{n+1}, copy it somewhere the next layer will be able to find it.
4. Deletes CDI_n and k_n before handing control to layer $n+1$.

For the first layer, CDI_0 is a unique device secret (UDS), an immutable, per-device secret bound at the the time of manufacture. Access to UDS has to be disabled before switching to layer 1, but this initial hardware-based "erasure" operation is out of scope for this paper. Critical to the other stages of the protocol is the erasure of the current layer's key k_n and its secret bundle CDI_n, so that later stages cannot impersonate earlier ones.

3.3 Trusted Code

Library code presents several difficulties:

1. Low-level applications like a bootloader typically do not use the Rust standard library. Some of the features available in Verus's equivalent `vstd` library are unavailable due to dependencies on Rust crates like `std`, which is not used in low-level software like kernels or bootloaders. This forces us to re-implement features found in the `vstd` library. This problem is also encountered by VeriSMo [25].
2. Verus's `vstd` library isn't yet mature enough to handle all kinds of raw, low-level operations. This includes operations on arbitrary regions of memory.

We draw attention to the amount of trusted code in use. A proof of functional correctness establishes that some program actually does what its specification says it does. We inherit Verus's assumptions. Verus assumes that a subset of the semantics of the Rust language are correct as implemented by the compiler. The Verus "standard library" uses, but does not verify, the Rust standard library, meaning it assumes they are correct. Similarly, for our bootloader, we focus on verifying application code, not the library code that it depends on. There will typically be several layers of libraries in a well-developed codebase. Since verifying entire codebases takes substantial effort, we assume the correctness of a substantial portion of library code. However, drawing the line between verified and trusted code is straightforward in Verus, allowing for modular, gradual verification of code, easing meaningful exploration of verification targets without requiring up-front verification of all supporting code.

4 Implementation

We develop our verified bootloader in Verus inside the context of the M^3 platform [3], targeting the RISC-V ISA, using the gem5 simulation platform. We started with an existing bootloader written for the platform which implements the DICE protocol. At the end, the bootloader finally loads and launches the kernel, handing control of the system over to it. The M^3 platform is a hybrid hardware/software platform which composes "tiles" using a secure network-on-chip interface. While M^3 is a custom platform, the tiles themselves can be commodity. For instance, a recent published version uses the open source and popular Rocket and BOOM cores [2]. We point this out to demonstrate that our bootloader is not especially tightly-coupled to the M^3 platform, and that with only minor adjustments it could be deployed to other RISC-V platforms as well. The bootloader stage binaries must be small enough to fit into on-chip SRAM to improve spatial isolation. Our choice of the M^3 platform is both convenient but deliberate: other works have argued Verus's suitability for works developed with "verification as a target," [5] which our bootloader was not.

Developing these specifications in Verus mostly takes the form of Hoare logic-style pre- and post-conditions which "decorate" functions, with additional individual assertion requests made to the SMT solver, which are occasionally needed because of one of: program structure, to improve proof performance, or guide the solver when it cannot automatically prove a property, a known theoretical limitation of SMT solvers. An example of such a decorated function is given in Figure 1:

Our implementation in Rust makes frequent use of raw pointers. To verify this code we used Verus's support for raw pointers which uses "ghost" types to leverage Rust's type and ownership checker to attach a notion of ownership and permissions to raw pointers. These ghost types are erased before reaching the Rust compiler, meaning they are fully consumed by Verus and generate no true Rust code. In our case, most work was spent decorating existing functions with pre- and post-conditions, splitting functions into smaller ones to make them easier to specify.

Verus's implementation of raw pointers suggests a global allocator which ensures pointer uniqueness and the absence of double-allocations. The allocator is considered trusted and should reasonably restrict the pointers that it grants. However, the existing allocation function in Verus's library uses Rust's underlying core allocator, which is not compatible with our bootloader. Developing our own allocator is

```
#[inline(always)]
#[cfg(target_arch = "riscv64")]
unsafe fn prepare_switch<Data: CtxData>(
    ctx: LayerCtx<Data>,
    where_to: *mut LayerCtx<Data>,
    Tracked(where_to_perm):
        Tracked<&mut PointsTo<LayerCtx<Data>>>,
    eclear: *const u8,
) -> (entry: usize)
    requires
        where_to@.addr == crate::MEM_OFFSET,
        eclear@.addr != 0,
        eclear@.addr >= where_to@.addr,
        old(where_to_perm).ptr() == where_to,
        old(where_to_perm).is_uninit(),
    ensures
        entry != 0,
        entry >= eclear as usize,
        where_to_perm.opt_value() ==
           MemContents::Init(ctx),
        crate::range_cleared(crate::MEM_OFFSET as int,
                             eclear@.addr as int),
{ ... }
```

Listing 1. A function's behavior is described in Verus with `requires` and `ensures` clauses.

not theoretically difficult, since no de-allocations take place and its only job would be to ensure uniqueness and the absence of pointer overlaps. However, support for access to raw memory is currently limited to *sized object* types, not to memory extents. Concretely, Verus currently supports a pointer to some struct (`*const myStruct`), but *not* to variable-sized byte arrays. Such support is necessary to implement even a simple allocator which works within a stage to prevent double-allocations. Extending Verus with this support is ongoing by the Verus authors and considered a matter of engineering, but was not available at the time. Thus we have no guarantee of pointer uniqueness at the proof level. Finally, the `range_cleared` property is attached to underlying Rust and Verus library functions, which we assume the correctness of.

To correctly control program flow at the point of clearing and program switching, we rely on one inline assembly routine that is trusted to uphold its specification. These trusted assembly routines are 53 lines long in total, or only2.5% of the total lines of code. Used during final stage-switching after the next stage's context has been copied to the next stage, this assembly routine clears the stack, clears the registers, and jumps to the next program binary. Inline assembly is required both to formally avoid undefined behavior and practically to prevent the compiler from inserting function calls which cannot return because the stack is cleared. Verifying assembly is a hard problem, with previous touchstone works also assuming its correctness [16].

The key property we aim for is the *secure erasure* of the current stage's context before handoff to the next one, as shown in Figure 2. To this end, we define its well-formedness property in Figure 2:

```
pub uninterp spec fn addr_cleared(a: int) -> bool;

pub open spec fn range_cleared(start: int, end: int) -> bool {
  forall|i: int| start <= i < end ==> #[trigger] addr_cleared(i)
}

pub open spec fn wf(layer : Bootstage) -> bool {
  range_cleared(layer.begin, layer.end)
}
```

Listing 2. Well-formedness property of a bootlayer.

The memory addresses between the beginning of memory (`layer.begin`) until a special linker-defined eclear symbol (`layer.end`) must be cleared, where this symbol points to an address at least after all data sections mentioned by the binary's ELF. In other words, we clear all the program's data. We assume that `eclear` is set correctly by the linker. The bootloader binaries have no heap, so allocations are stack-based and data is either on the stack, in the `.bss` section, or explicitly copied somewhere. For clarity, the function in Listing 1 shows that `range_cleared` is true to a higher function that ensures that `wf` is true for the whole bootlayer.

Copying the context to its next position and clearing the program data are both required to satisfy our well-formedness property, but they happen in two separate places: one in ordinary Rust code, the other in the final inline assembly snippet. However, specifying the property in the correct way allows the SMT solver to deduce that because the two functions clear two separate, but adjacent ranges, this is equivalent to the overall property. This does not require substantial proof effort and shows that meaningful and important properties can be shown just from *specifying* the code using modern tools.

4.1 Trusted Code

Rust core library	59857
M^3 base library	8931
Verus `vstd` library	19598
Trusted assembly routines	53

Table 2. Trusted Lines of Code in our implementation

As mentioned in the design section, we trust several libraries and code snippets: the underlying Rust `core` library, the M^3 `base` library, the Verus `vstd` codebase, and the final trusted assembly routines. The numbers given in Table 2 represent the raw lines of code contained in each source; for the libraries, only a very small subset of all features are used, and thus the raw number is a *ceiling* on the effective trusted lines of code from that source. As these numbers show, trusted libraries significantly inflate the TCB of even a simple project like the bootloader.

5 Evaluation & Discussion

We evaluate our work in terms of the difficulties experienced while using Verus and the performance overheads that Verus brings. We argue that Verus is lacking expressivity and quality-of-life features, but that one can nevertheless develop meaningful proofs in it with proof-to-code ratios of as low as 0.56, when targeting simple properties of real-world programs.

5.1 Verus

Verus is relatively new research software and does not (and does not aim to) support all possible Rust features. This immaturity still presents difficulties in using Verus. We describe some of these difficulties, identify deficiencies with Verus, and describe what support is needed to resolve these difficulties.

5.1.1 Retrofitting. To begin with Verus, one adds pre- and post-conditions to functions, "importing" code into Verus. Trusted code is moved out of the verification path and claims are given and solved by specification and any proof functions, respectively. This initial step of "importing" code imposes a substantial burden up-front, requiring, as the first step, the programmer to write substantial boilerplate to mark unrelated code or library code as trusted. Unlike the actual specification of the program, this code *could* be generated automatically. For medium to large codebases such a utility would significantly reduce Verus "on-boarding" times.

5.1.2 Developer experience. Verus is based off Rust syntax, but is different enough to sometimes pose problems. Syntax and semantics such as `ghost` and `tracked` variables are poorly defined, as Verus is relatively new, and correct usage of these concepts is inferred mostly from Verus code itself.

On the other hand, developing proofs using Verus is intuitive and *feels like* writing Rust. There's a "closed loop" which roughly begins by improving the specification of a function to better describe its properties, which then causes the solver to fail to prove something about the code. It is straightforward to then insert stand-in assumptions which repair verification, and these stand-in assumptions point the developer to where the proof code is lacking or specifications that are not clear enough for the SMT solver to automatically deduce the fact. We began this work with no prior experience in verification, showing Verus's accessibilty for non-experts.

The supported subset of Rust is documented and is generally sufficient. However, we found in several instances that missing support required re-engineering to work around. Notably, Verus currently does not support associated constants, an example of which is shown in Figure 3, and Verus has only partial support for static items. These language constructs encourage explicit, structured programming and working around these limitations increased code verbosity and harmed readability. Concretely, the lack of associated constants means that some data associated with each layer must be encoded as a *non-const* function which always returns the same value. In practice, the compiler optimizes out the function call, but explicitly communicating intent should be preferred when possible. In another case, we downgraded certain function logic from generic to specific. While an experienced Verus user likely understands how to structure code to avoid these limitations, it makes adopting Verus into an existing codebase more difficult. We emphasize that these remarks are not criticisms of Verus's design, but only on the state of development of Verus.

```
struct Foo { field : i32 }
impl Foo {
    // associated constant
    const MAGIC : usize = 0xdeadbeef;
    // associated constant fn
    const fn magic () -> usize { 0xdeadbeef }
}
```

Listing 3. Example of code currently unsupported by Verus

5.1.3 Opportunities for Improvement.

We have identified these areas for improvement to Verus, based on our difficulties:

- Automated generation of trusted code stubs.
- Improved documentation of Verus features.
- Improved support for reasoning about untyped, raw memory ranges.
- Supporting a greater subset of Rust language semantics.

5.2 Code Size

Our primary concerns for code size are the overheads that using Verus has on the final bootloader binaries and the extra developer effort required for verification. Much of the extra code required by Verus is ghost code erased before Rust compilation. However, our source code required some re-arrangement to work with Verus. In principle, this re-arrangement can result in changed binary sizes, slowdowns, or other performance losses.

In the cases of the sizes of two of the three binaries, shown in Table 3, our changes to resulted in no meaningful change. Surprisingly, in the case of the second stage, Blau, adding Verus resulted in a *smaller* binary. Inspection of the binaries reveals that the Verus code results in better optimization by the compiler. This is not unexpected, however, since our initial codebase has *not* been aggressively optimized for compactness. As a reminder, Brom is the first stage, Blau the second, and Rosa the last stage which performs this layer switch.

Stage	Orig.	With Verus
Brom	30K	30K
Blau	58K	42K
Rosa	109K	110K

Table 3. Binary sizes are largely unchanged after applying to Verus.

	Application	Library	Total
LoC Without Verus	725	607	1,332
LoC With Verus	899	1,175	2,074
Proof LoC	*174*	*568*	*742*
Proof Ratio	0.24	0.94	0.56

Table 4. Verus can make the proof burden for a simple task quite low.

A classic measurement of software verification evaluations is lines of proof code required. We used Verus's included code formatter. Shown in Table 4, we note that the proof ratio is quite low at 0.56, and much smaller for the core "application code" than for the library code. This is because the interfaces from verified to trusted code are in the library. In contrast, the application code itself is mostly pre- and post-condition specifications and specification functions describing program properties. Few manual proof functions were required in our case, as the SMT solver used by Verus, Z3, can deduce many facts/properties on its own. Proof functions were often eliminated once it was clear *how* to structure our specifications to guide the solver. Overall, the added code was split evenly between pre- and post-conditions and assumptions to wrap trusted library code. Our final proof ratio was 0.56, a very positive but also optimistic result, which we expect to worsen as more properties are incorporated into the specification. However, this supports the claim of Verus as a "lean" verification framework.

5.3 Performance

Finally, we measured the execution of the binaries in gem5 to characterize how much latency was introduced by the changes to code organization, shown in Table 5. These results show no compelling performance loss or gain overall. Although a difference in terms of percentage, Brom's apparent performance improvement is on the order of a few hundred microseconds against an overall program duration of less than two thousand microseconds. These improvements are again likely due to compiler optimizations, since the bootloader code is not optimized for performance. Indeed, even slight changes to the code result in significant layout changes by the compiler which influences these results, since they are small.

Stage	Orig.	With Verus	Diff.	Percent
Brom	1.8	1.5	-0.3	-16.78%
Blau	52,977	52,943	-34	-0.06%
Rosa	72,205	76,958	4,753	6.58%

Table 5. Verus does not add meaningful latency to our boot process. Numbers given in microseconds (μs).

5.4 Generalizability to Other Software

We believe our specification, although narrow, is a reasonable exemplar of problems in the domain of bootloaders and system software. Modeling the interaction between hardware and software, especially the correct handling of values at or below language level is a problem other works have dealt with: SeL4 chooses to not verify boot code or context switching. A significant part of VeriSMo is dedicated to representing the interaction of memory hardware. We do not claim to go beyond what previous works have done in this regard, but point them out to suggest that this type of property has high relevance in this software domain.

6 Future Work and Conclusion

Future work directions include better control over pointer provenance through some kind of allocator, as well as a no-copy property which ensures the per-layer secrets are also *not copied* anywhere else.

We use Verus to verify that a key property of our TEE bootloader holds, under some assumptions. We specifically use an existing codebase and show that such an approach is feasible for projects of moderate complexity because it is possible to "fence off" Verus code from the rest of the codebase. We show that Verus can be used for cheap, performant verification, even in the hands of those that did not develop the tool. Further, we provide a first attempt to apply Verus to software at the lowest levels of abstraction, which have historically been difficult to prove. Our work sets the stage for extending our specification, not only to capture other aspects of the DICE protocol, but also its role as a bootloader in general.

Acknowledgments

This research was co-financed by public funding of the state of Saxony/Germany and by the European Union's Horizon Europe research and innovation program under grant agreement No. 101094218 (CYMEDSEC).

References

[1] Ashish Agarwala, Priyanka Singh, and Pradeep K Atrey. 2017. DICE: A dual integrity convergent encryption protocol for client side secure data deduplication. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2176–2181.

[2] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. 2022. Efficient and scalable core multiplexing with M^{3}v. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 452–466. doi:10.1145/3503222.3507741

[3] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 189–203. doi:10.1145/2872362.2872371

[4] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings* (Pasadena, CA, USA). Springer-Verlag, Berlin, Heidelberg, 88–108. doi:10.1007/978-3-031-06773-0_5

[5] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A hybrid approach to semi-automated Rust verification. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 970–992.

[6] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 21)*. 295–312.

[7] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Gancher, Milijana Surbatovich, Bryan Parno, and Reviewing Model. 2025. Vest: Verified, secure, high-performance parsing and serialization for Rust. In *Proceedings of the USENIX Security Symposium.*

[8] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. 2023. Atmosphere: Towards Practical Verified Kernels in Rust. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification* (Koblenz, Germany) *(KISV '23)*. Association for Computing Machinery, New York, NY, USA, 9–17. doi:10.1145/3625275.3625401

[9] Xiangdong Chen, Zhaofeng Li, Jerry Zhang, and Anton Burtsev. 2024. Veld: Verified Linux Drivers. In *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification* (Austin, TX, USA) *(KISV '24)*. Association for Computing Machinery, New York, NY, USA, 23–30. doi:10.1145/3698576.3698766

[10] Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno. 2024. A Framework for Debugging Automated Program Verification Proofs via Proof Actions. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 348–361.

[11] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 90–105.

[12] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (June 2024), 25 pages. doi:10.1145/3656422

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[14] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596

[17] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[18] Hayley LeBlanc, Jay Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. 2025. PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, USENIX, 839–857. https://www.microsoft.com/en-us/research/publication/power-never-corrupts-tool-agnostic-verification-of-crash-consistency-and-corruption-detection/

[19] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.

[20] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[21] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V Thakur. 2021. DICE*: A formally verified implementation of DICE measured boot. In *30th USENIX Security Symposium (USENIX Security 21)*. 1091–1107.

[22] The Coq Development Team. 2024. *The Coq Proof Assistant*. doi:10.5281/zenodo.14542673

[23] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 321–330. doi:10.1145/3510457.3513031

[24] Yi Zhou, Amar Shah, Zhengyao Lin, Marijn Heule, and Bryan Parno. 2025. Cazamariposas: Automated instability debugging in smt-based program verification. In *Conference on Automated Deduction*, Vol. 122.

[25] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A Verified Security Module for Confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 599–614. https://www.usenix.org/conference/osdi24/presentation/zhou