All Your Assembly Belongs to Rust

Automated Lifting for Uniform Testing and Verification

Charly Castes
EPFL
Switzerland

Gurvan Debaussart
EPFL
Switzerland

Thomas Bourgeat
EPFL
Switzerland

Abstract

Inline assembly in systems software is often a source of challenges, a potential source of undefined behavior, and largely opaque to verification and bug-finding tools. The standard approach to verifying such code axiomatizes instruction semantics in the program logic, requiring custom models in the logic and specialized tooling.

In this paper, we explore an alternative: to translate Rust code containing RISC-V inline assembly into pure Rust code by *emulating* each instruction using a machine model extracted from the official RISC-V Sail ISA specification. The resulting code can then be processed by standard Rust verification and testing tools: bounded model checkers (Kani), undefined-behavior detectors (Miri), and fuzzers, with no custom axiomatization of instruction semantics. We go over the different challenges raised by our approach: pure register operations, memory accesses, control flow, and system instructions, and show how each category is handled by the translation. We have applied this approach to verify inline assembly in two systems: Miralis, a RISC-V firmware hypervisor, and a dynamic root-of-trust firmware.

ACM Reference Format:
Charly Castes, Gurvan Debaussart, and Thomas Bourgeat. 2026. All Your Assembly Belongs to Rust: Automated Lifting for Uniform Testing and Verification. In *14th Workshop on Programming Languages and Operating Systems (PLOS '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3831586.3838148

1 Introduction

Firmware, hypervisors, and operating system kernels rely on inline assembly for operations that cannot be expressed in a high-level language: configuring physical memory protections, setting trap handlers, performing context switches, and accessing memory-mapped devices. Outside of systems code, inline assembly also appears in cryptographic libraries (for constant-time guarantees) and high-performance numerical code (for using vector instructions).

This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2877-8/2026/09
https://doi.org/10.1145/3831586.3838148

These assembly fragments are a persistent blind spot for verification and bug-finding tools. Tools such as ASan [16, 24], Valgrind [19], and Miri [11] cannot reason about the effects of inline assembly on high-level program state, and programmers are mainly on their own. This is because such tools reason about or instrument code at the source level. When verification is attempted, standard approaches fall into one of two categories. The first is to *axiomatize* the effects of each instruction in program logic [14, 17], typically one instruction at a time. This requires hand-written, architecture-specific models expressed in the logic. The second is to perform a *lowering*, compiling down to binary, and then reason about the whole system at the assembly level [8, 18]. This requires dedicated tools and formal CPU models, while losing access to high-level language semantics and high-quality tooling. Further, cross-compilation is not possible: testing RISC-V assembly on an x86 development machine requires a full-system emulator.

We propose a different direction. Rather than lowering mixed-language code to assembly for verification, we *lift* the assembly into the host language [15, 21]. Concretely, we automatically replace each inline assembly instruction with a call to an executable machine model compiled from the official RISC-V Sail ISA specification [1], producing a pure Rust program with identical observable effects on the Rust Abstract Machine. This pure Rust program can then be fed to any tool in the Rust ecosystem: Kani for bounded model checking, Miri for undefined-behavior detection, or standard fuzzers for property-based testing. No custom axiomatization is needed; the instruction semantics come directly from the official Sail specification. While it is no replacement for typical QEMU-based testing, our approach yields a lightweight cross-platform execution path for programs that contain only a modest amount of inline assembly: a relatively common case in systems software.

This idea is grounded in a principle articulated by the Rust community [9]: for an `asm!` block to be well-defined, there *must* exist a pure Rust program that performs the same state transformation on the Rust Abstract Machine. Our approach simply makes that witness program explicit; see Section 2.

The translation needs to be careful with many subtleties: assembly that modifies the current execution environment, such as changing privilege levels or updating the active page tables, falls outside the scope of automatic translation. Nonetheless, our approach covers a significant portion of

assembly use cases. We explore these boundaries by categorizing inline assembly along four dimensions: pure register operations, memory accesses, control flow, and system instructions. We highlight the challenges of each category, and show how we handled them in Section 3.

We have applied this approach to two RISC-V systems software projects. We extended the verification of Miralis [5] to include all inline assembly snippets, which previous efforts did not cover. In a new dynamic root-of-trust firmware (~600 LOC), we used our translation approach to prove properties over reachable system state, such as correct PMP configuration enforcing the desired region isolation. These new efforts are presented in Section 4.

2 Background

Inline assembly appears throughout the software stack for several reasons:

System instructions: Configuring physical memory protections (PMP), setting trap handlers, reading hardware identifiers (*e.g.,* `mhartid`), and manipulating privilege levels all require instructions with no high-level language equivalent.

Unusual control flow: Context switches save and restore register state, transferring control between execution contexts in ways that violate the assumptions of structured programming.

Performance and security: Cryptographic code often uses hand-written branchless assembly to avoid timing side channels. Similarly, numerical libraries can often make use of vector instructions for performance. Prior work [21] has shown that a surprisingly large number of regular user-space applications have at least one dependency that makes use of inline assembly (in C++).

Yet, testing and debugging these assembly fragments is difficult. Tools like ASan and Valgrind often treat inline assembly as opaque.

2.1 Inline Assembly in Rust

Rust's `asm!` macro provides a structured contract-like interface to inline assembly. The programmer declares which registers are inputs, outputs, and clobbers. Based on these declarations and without further checking, the compiler interleaves compiled Rust code with user-provided assembler. Any mismatch between the manual declarations and assembly behavior results in Undefined Behavior (UB).

A key principle of inline assembly in Rust is that for `asm!` to be well-defined, *there must exist a pure Rust program that performs the same state transformation on the Rust Abstract Machine.* For instance, when assembly is used as an optimization, such as vector instructions, there must exist a pure Rust program that produces the same effect. In the case of system instructions, assembly must produce no side effects observable from the Rust Abstract Machine. If that principle is not respected, the inline assembly is UB and the compiler can make arbitrary assumptions about the program's behavior.

In Figure 1 we illustrate an abstract formalization of this principle for well-defined inline assembly. The top row shows the Rust Abstract Machine evolving through states s_0, s_1, s_2, s_3; the `asm!` block is a single opaque step from s_1 to s_2. The bottom row shows the *witness*: a sequence of pure Rust steps that performs the same state transformation by emulating each assembly instruction on a cross-product state (s, pm) pairing the Rust state s with the emulated physical machine state pm. The dashed vertical arrows represent a simulation relation: the Rust-visible projection of the witness must match the original execution at the entry and exit of the `asm!` block. Our approach constructs this witness explicitly, leveraging the official RISC-V Sail ISA specification as the source of the side state type $pm_0, pm' \cdots pm_1$ and the instruction semantics acting on it.

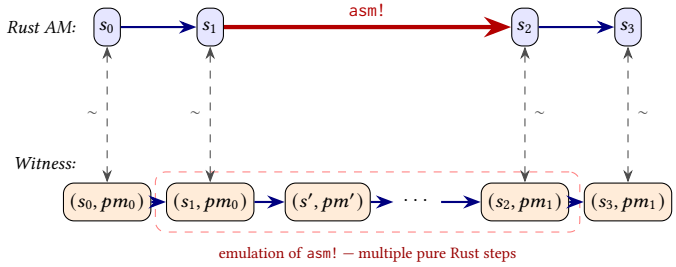

Figure 1. The `asm!` block (top, red arrow) is a single opaque step on the Rust Abstract Machine. The witness (bottom, dashed box) emulates it as a sequence of pure Rust steps over a cross-product state pairing Rust state with physical machine state.

2.2 The Sail ISA Specification Language

Sail [2] is a language for describing instruction-set architectures. The RISC-V community maintains a RISC-V ISA specification written in Sail that serves as the reference definition of instruction semantics. Similarly, the official ARM ISA specification is written in ASL [22], another ISA specification language, with automated translations to Sail maintained by the community. Sail models can be compiled to C, OCaml, Lean, and other targets to produce reference simulators.

Our approach relies on a Sail-to-Rust compiler, i.e., a new backend for the Sail compiler that translates the Sail specification into a Rust library exposing the functions of the ISA semantics: execute, decode, etc. This library is the foundation of our executable machine model: calling a function with the appropriate register operands produces the same state transformation as executing the corresponding instruction on hardware.

2.3 Testing and Verification Tools for Rust

The Rust ecosystem contains many tools to gain trust in our software. These tools vary in how much user input they

require to get working and how much trust they allow us to gain in our system.

Unit testing: The main tool for unit testing in Rust is `cargo test`, which is part of the standard Rust distribution. Unit tests are expected to run as user-space programs on the host architecture. Unit tests involving inline assembly for a different architecture or using system instructions must be cross-compiled and run inside an emulator.

Bounded model checking and symbolic execution: A way to gain even more confidence is to test all possible inputs up to a certain bound. The primary tools in this space are Kani [6, 25] and Soteria [28], which can prove properties about Rust programs or find counterexamples. None of them currently support inline assembly.

Dynamic Undefined Behavior detection: Instead of manually writing the expected behavior of our programs, we may also be interested in knowing whether our programs contain *undefined behaviors*, such as invalid pointer use, data races or alignment violations. Tools such as Miri [11] can execute programs step-by-step on concrete inputs using a strict virtual machine and search for these undefined behaviors that native execution might silently ignore. Miri does not currently support inline assembly.

Fuzzers: To find undefined behaviors and unexpected behaviors, we could also test our program with pseudo-random inputs. Tools such as `cargo-fuzz` do just that. Just like unit tests, they can only be used with user-space assembly via cross-compilation.

Deductive verification: To gain the highest possible level of trust in our systems, we can annotate our programs with formal specifications. We can for example annotate our programs with pre- and post-conditions, and then prove that any call to the function, assuming the pre-condition, ensures that the post-condition is true after the function execution. Primary tools in this space are Verus [13] and Creusot [7], neither of which currently supports inline assembly.

As can be noted, most if not all of these tools operate on pure Rust code and cannot reason about inline assembly directly. The goal of our translation is to bridge this gap by producing pure Rust that these tools can process and analyze, therefore allowing us to extend the support of existing verification and testing tools from the Rust ecosystem to code containing inline assembly.

3 Lifting Assembly to Pure Rust

In order to enable the use of existing Rust tooling on code containing inline assembly, we introduce a new `soft_asm!` macro compatible with Rust's built-in `asm!` macro. Based on the compilation target, `soft_asm!` either transparently forwards the assembly to `asm!`, or *lifts* assembly into pure Rust code that emulates the assembly snippets.

In emulation mode, our macro replaces each instruction with a call to a trusted machine model, automatically derived from the official RISC-V Sail ISA specification compiled into Rust. After macro expansion, our `soft_asm!` macro leaves no built-in Rust `asm!` macros, and can therefore be compiled to any platform and processed by any Rust tool.

```rust
pub fn execute(&mut self, instr: ast) -> Trap;
pub fn get(&mut self, reg: GeneralRegister) -> u64;
pub fn set(&mut self, reg: GeneralRegister, value: u64)
```

Figure 2. Signature of some functions of the model.

In this section, we first go over how the CPU is modeled in Rust, and then categorize different kinds of inline assembly, incrementally highlighting the purpose and difficulties of using each category (summarized in Table 1), as well as how our tool handles them.

What ASM does	Challenges
Pure register ops	Register allocation and non-determinism
Memory accesses	Unsafe dereference, provenance
Control flow	Decompilation, Rust interop
System instructions	Traps, effect on Rust Abstract Machine

Table 1. Inline assembly categories and their challenges.

3.1 A Rust CPU Model

The CPU model is compiled from the Sail specification as a Rust structure which contains all registers of the used part of the model, along with an extra configuration field for supported RISC-V extensions. In total, the model accounts for 7.4k lines of automatically generated Rust code, presented as a library that can be configured with the desired RISC-V extensions by user code. We use a custom Sail compiler back-end, adapted from the original effort to verify the Miralis virtual firmware monitor [4], to support most of the base RISC-V ISA. The `soft_asm!` macro can access the CPU through a thread-local `core` variable that exposes methods to access registers or execute instructions (Figure 2).

3.2 Pure Register Operations

The simplest case is a sequence of arithmetic and logic instructions over declared input/output registers, with no memory access, branching, or system state. The assembly block acts as a pure function mapping outputs deterministically from inputs. Pure register operations are common when working with vector instructions, as well as in constant-time cryptographic code, where branches on secrets are forbidden to prevent timing side channels. To understand the translation process, let's look at a toy branchless absolute value computation using three instructions: `srai`, `xor`, `sub` (Figure 3).

Translation: When expanding in emulation mode, our `soft_asm!` macro replaces the assembly snippet with a pure Rust code block invoking methods on the `core` variable that emulates a physical CPU core (Figure 4).

```rust
let mut result: u64 = 0;
let mut sign: u64 = 0;
soft_asm!(
    "srai {sign},   {x}, 63",
    "xor  {result}, {x}, {sign}",
    "sub  {result}, {result}, {sign}",
    x      = in(reg)    x as u64,
    sign   = out(reg) sign,
    result = out(reg) result,
);
```

Figure 3. Absolute value computation using inline assembly.

```rust
let mut result: u64 = 0;
let mut sign: u64 = 0;
{
    core.set(reg::X1, x as u64);
    match core.execute(
      ast::SHIFTIOP(reg::X1,63,reg::X2,sop::SRAI)
    ) { ... };
    match core.execute(
      ast::RTYPE((reg::X1,reg::X2,reg::X3,rop::XOR))
    ) { ... };
    match core.execute(
      ast::RTYPE((reg::X2,reg::X3,reg::X3,rop::SUB))
    ) { ... };
    sign   = core.get(reg::X2);
    result = core.get(reg::X3);
};
```

Figure 4. Absolute value computation in pure Rust.

The translations above and in the rest of this paper are slightly simplified for the sake of clarity, but representative of the actual output. The `core` object corresponds to the *pm* state in Figure 1. It holds the full architectural state (registers, CSRs, privilege mode), as generated by the Sail-to-Rust compiler. The preamble on line 4 in Figure 4 loads the Rust input `x` into an emulated register via `core.set`, as specified in line 7 of Figure 3. Each instruction becomes a call to `core.execute` with the decoded AST node from the Sail specification. The postamble, on lines 14 and 15 in Figure 4, extracts the result via `core.get` to the variable specified on lines 8 and 9 of Figure 3. Note that traps are checked for every instruction, for simplicity. A smarter `soft_asm!` macro could analyze the RISC-V model to prune trap checks for instructions that never trap.

Even in this simple case, this kind of translation enables catching assembly-level bugs: the inline assembly can now be tested for correctness using standard tools, for instance proving equivalence to `if x < 0 { x.wrapping_neg() } else { x }` for all 64-bit inputs using Kani.

In Figure 3, notice that the assembly block does not contain an explicit mapping between machine registers and Rust variables. The inline assembly contract only associates some Rust variable with some abstract register, and the actual register allocation is left to the compiler. Our Rust macro therefore picks one particular register allocation: in our example, the variable `x` is placed in register `X1`, `sign` into `X2`, and `result` into `X3`. Unfortunately, this means that tests and verification might miss bugs that materialize only with a specific register allocation.

```rust
let rsv1 = core.get(reg::RS1);
let rvd = unsafe { *(rsv1 as *const u64) };
core.set(reg::RD, rvd);
```

Figure 5. Translation of `ld rd, 0(rs1)`.

Yet, our approach enables catching bugs with a fixed register allocator, as accessing any register not specified in the inline assembly contract is undefined behavior. To catch reads from undeclared input registers, the test or verification harness can initialize all registers to nondeterministic values (*e.g.*, using `kani::any()` under Kani, or random values under fuzzing). Similarly, after execution, the harness can assert that all registers not declared as outputs retain their initial (garbage) value.

3.3 Memory Accesses

Another important category of instructions is loads and stores. Notice that the memory is not part of the explicit machine state struct, and so must instead be accessed by finding the appropriate Rust object in the Rust Abstract Machine, via Rust dereferences.

Translation: A load instruction, *e.g.*, `ld rd, 0(rs1)`, translates as follows: the base address is first read from the emulated register file via `core.get(rs1)`, then the actual memory access is performed as an unsafe Rust dereference, and finally the loaded value is written back into the emulated register file via `core.set(rd, ...)` (Figure 5).

This is a valid translation that emits unsafe but runnable code. In particular, during concrete execution, the program crashes on invalid addresses. Further, tools such as Miri and Kani check raw pointer dereferences for UB, according to a given borrowing model [10, 26], and will flag memory accesses that point outside a live Rust allocation or violate aliasing or alignment requirements. Tools will also flag bogus accesses to struct members via byte offset in case of offset miscalculation or missing `#[repr(C)]` annotations (required in Rust to force a specified memory layout). The stack is treated like any memory, with the additional constraint that the stack pointer must be restored at the end of the assembly block.

This translation is limited when tackling more complex instructions such as vector load and store instructions, which support precise loading mechanisms with masks and strides. A future approach would be to model the memory load and store of the Sail model with user-provided Rust functions. This would allow defining custom memory controller functions, possibly emulating memory-mapped I/O.

3.4 Control Flow

Contrary to Rust, assembly supports unstructured control flow. Therefore, the macro performs a lightweight decompilation pass to reconstruct high-level control flow: it builds a

```rust
match core.execute(
  ast::CSRReg((bv(0), reg::X1, reg::X0, csrop::CSRRW))
) {
  Trap::Some(addr) => handle_trap(addr, &[handler1, ...]),
  Trap::None => (),
};
```

Figure 6. Handling traps in exceptional control flow.

control-flow graph of the assembly block, identifies branch targets, and emits a structured Rust control flow when it finds a known pattern (`if`, for example). As Rust's control flow is more restrictive, the macro raises an error if it encounters unsupported patterns.

Translating structured control flow: Constructions such as if branches and simple loops are recovered and translated into equivalent Rust statements. More general control flow, such as state machines, could be recovered through traditional decompilation techniques, such as the *relooper* algorithm [27], but is not necessary for our applications and is therefore left as future work. Note that for a Rust translation to exist, the assembly control flow should never leave the inline assembly snippet, except for exceptional control flow and function calls, as discussed below.

Translating exceptional control flow: Assembly code sometimes triggers exceptional control flow, which we model explicitly in the translation by letting the user provide a list of expected trap handlers. Each call to `core.execute` returns either a successful execution or a trapping address. In case of a trap, the address is checked against the provided list of handlers. The matching trap handler is invoked, or a Rust panic is raised if none is found. Note that the trap handler must have been installed in the appropriate register before, either by the test harness, or by a previous assembly snippet. For instance, accessing the undefined CSR number 0 will reach the `Trap::Some` branch in Figure 6.

Translating function calls: Finally, assembly code might need to call into Rust code. The macro recognizes the `call <target>` pseudo-instruction (translated to `jal ra, <target>`) and translates it to a direct function call following the platform's ABI. Because Rust macros do not get access to type-checking information, our macro cannot infer the function ABI on its own. Instead, we require explicit user annotation to provide the function ABI, and emit code that fails type-checking if the user provides the wrong ABI. For instance, calling the `add_two` function in Figure 7 gets translated to the code shown in Figure 8.

3.5 System Instructions

Finally, in the context of low-level system code, inline assembly is used to read and write system registers. This category can add side effects on machine state beyond the declared `asm!` interface.

Translation: As system instructions are already part of the Sail specification, no further changes to the translation

```rust
extern "C" fn add_two(a: u64, b: u64) -> u64 {
    a + b
}
let result: u64;
soft_asm!(
    "// #[abi(\"C\", 2, u64)]",
    "call {func}",
    func = sym add_two,
    result = inout("a0") 100 => result,
    in("a1") 42,
);
```

Figure 7. Function call annotation in `soft_asm!`.

```rust
core.set(reg::X10, 100 as u64);
core.set(reg::X11, 42 as u64);
let arg0 = core.get(reg::X10);
let arg1 = core.get(reg::X11);
let _: extern "C" fn(_, _) -> u64 = add_two; // Typecheck
let ret_val = add_two(
    FromRegister::from_register(arg0),
    FromRegister::from_register(arg1),
);
core.set(reg::X10, ret_val as u64);
result = core.get(reg::X10);
```

Figure 8. Translation of the function call from Figure 7.

are needed. The emulated machine state includes CSRs, privilege mode, and other architectural states. System instructions simply read and write fields of this struct.

In the context of system instructions, typically the user might be interested in asserting properties about *virtual machine state* instead of just properties about original Rust variables. For example, one will typically be interested in asserts of the form: Does the assembly block trap unexpectedly? Is the PMP configured correctly after this block? Are interrupts enabled at the expected program points? The translation allows answering all those questions, as they rely on querying the internal CPU state. Because `core` is a Rust variable, it can be accessed by test and verification harnesses to assess the desired properties.

The translation process by itself does not automatically enable existing tools to find all bugs linked to system instructions. Indeed, while assembly snippets will execute faithfully, the impact on Rust code might not be captured. For instance, disabling floating-point instructions from an assembly snippet will correctly cause floating-point instructions to trap in assembly code, but not floating-point operations in pure Rust code. Similarly, as a consequence of our translation of loads and stores to direct pointer accesses, the MMU subsystem is bypassed by memory accesses both in assembly and pure Rust code. It remains possible to use our `soft_asm!` macro to verify properties about floating points or the MMU, but those must be explicitly encoded in the test harnesses.

4 Evaluation

In this section, we report our experience using our `soft_asm!` macro in two contexts: extending the scope of formal

verification of Miralis' firmware virtualization subsystem, and developing a new dynamic root-of-trust firmware.

4.1 Virtual Firmware Monitor

The virtualization subsystem of Miralis has been verified for correctness using Kani [5], but previous verification efforts only covered the Rust code. The Miralis codebase centralizes all assembly snippets into a single module. The original verification replaced that module with a *manually written* pure Rust version.

As an experiment for the applicability of our approach, we removed the mock module used for verification, and replaced all `asm!` with `soft_asm!` in the original assembly module. On a total of 208 lines of inline assembly, switching to `soft_asm!` required manually updating 6 lines due to differences between `soft_asm!` and Rust's `asm!`. One explicit call annotation, two explicit type casts, an explicit mention of a global symbol, and two disambiguations between `add` and `addi` were needed. Re-running the tests and verification surfaced two new bugs that were not detected by previous methods: a register that should have been declared as clobbered, and one bit lost during CSR manipulation due to a hardwired 0.

4.2 Dynamic Root-of-Trust Firmware

We used `soft_asm!` while developing Flashpoint [12], a new dynamic root-of-trust firmware in order to verify functional correctness and security properties through exhaustive model checking using Kani. The goal of the verification is to ensure that the root of trust correctly transitions the system through a set of well-defined states. In particular, the root of trust must be resilient to arbitrary initial system state, and leave the system in a state with particular memory protection configured.

The root-of-trust entry point is an assembly snippet that configures the Rust environment (*i.e.,* the stack) and calls into the Rust entry point. The Rust code itself contains inline assembly snippets, and eventually returns to the assembly entry point which sets up the environment for execution after the root of trust. The root of trust also depends on two new RISC-V instructions. To add support for the new instructions in `soft_asm!`, we extended the Sail model with the two new definitions, and re-compiled it into a custom Rust model.

The verification consists of two parts: CPU configuration and Rust-level assertions. We emulate arbitrary system state by setting all CPU registers (general-purpose and CSRs) to symbolic values in the verification harness. After execution, the verification harness runs both Rust-level asserts to check internal root of trust state, and asserts against the CPU model. If the verification succeeds, no initial CPU state can cause the root of trust to violate the assertions. For instance, we prove that the root of trust always configures memory protections correctly (through PMP) using a symbolic address `addr` and asserting that `core.pmp_check(addr)` returns true only on explicitly allowed memory regions according to our specification, where `pmp_check` is the function from the Sail model used to check valid memory accesses.

Complete model checking of our dynamic root of trust takes 3 hours and consumes ~50GB of memory for each of the six transitions to verify on an AMD EPYC 9224 machine. The significant verification time and memory usage are due to insufficiently aggressive dead-code pruning, resulting in SMT queries encoding the behavior of the whole RISC-V specification instead of the reachable subset. While fixing the root cause is left for future work, relying on Rust unit tests, Miri, and fuzzing has proven very helpful during development and final verification.

5 Related Work

Most existing approaches to verifying mixed-language code that includes assembly fall into one of two categories: reasoning at the assembly level, or axiomatizing assembly effects in a program logic.

Decompilation into logic: The dominant approach translates assembly into a verification logic and reasons there. In that spirit, TickTock [23] uses refinement types (via Flux) to specify contracts for assembly routines, with instruction effects modeled as trusted axioms in the type system. One key difference from our approach is that axiomatized models must be trusted and cannot themselves be tested or verified against a reference; in our approach, the model *is* the official ISA specification, and any discrepancy between the model and hardware is either a specification bug or a translator bug, but not a modeling error. Further, our approach produces pure Rust code that can be fed to any Rust tool and executes code on the host, whereas axiomatization typically requires working with dedicated tooling.

Assembly-level verification: Another approach is to reason entirely at the assembly level. Tools such as Serval [18] enable verification of compiled binaries, while tools such as Vale [3] enable direct verification of assembly. Such tools rely on custom-built machine models, and lose the ability to reason about high-level language concepts. Vx86 [15] proposes a different model, lifting the assembly code into C and then using C-level verifiers, but still relies on a custom axiomatized machine model to do so.

Decompilation into C: The idea of decompiling assembly into a high-level language for verification is not new. A closely related work, TINA [21], uses a decompiler to identify the snippets of inline assembly in the binary, transforms them into C, recompiles, and performs equivalence checking on the resulting binaries. TINA shares our concerns about inline assembly blocking the utilization of standard tools, but only targets user-mode assembly. In contrast, we frame the problem as an emulation problem, handle system state and devices, and use the official Sail ISA specification.

6 Future work

Deductive verification: While it is challenging on our extracted codebase without contract annotation, recent work [20] suggests that un-annotated code can still be used in verification when the code body *is* the specification, as in our case.

Memory: We plan to enhance the memory model to support seamless device emulation integration.

Other ISA: While we currently focus on RISC-V due to its mature and official Sail specification, our method should be portable across ISAs. For instance, there exists an official translation of the ARM ISA specification to Sail, and a formalization of the user-space fragment of x86; both could be supported by `soft_asm!`.

Other languages: Our approach is not specific to Rust; it benefits from the convenient Rust macro system, but could be applied to other systems languages such as C or C++ using a custom pre-processor, for instance.

7 Conclusion

In this paper, we propose to *lift* assembly code into pure Rust code to enable testing and verification using existing off-the-shelf tools on projects that include assembly. We propose a new `soft_asm!` macro, compatible with Rust's `asm!`, that translates assembly into pure Rust code. The resulting Rust code emulates assembly execution using an accurate CPU model compiled from the official RISC-V Sail specification to a Rust library. We integrated `soft_asm!` in Miralis, finding two new bugs and demonstrating good compatibility with `asm!`, and leveraged `soft_asm!` to verify a new dynamic root-of-trust firmware.

Artifact Availability

The code for the `soft_asm!` macro presented in this paper is open-source. An `asm!` macro compatible with Rust's built-in `core::arch::asm!` is provided as a Rust library under the `softcore-asm-rv64`[1] crate. The `softcore-asm-rv64` crate depends on the Rust CPU model, distributed as the `softcore-rv64`[2] crate. The `soft_asm!` macro is a thin declarative macro that wraps the `softcore_asm_rv64::asm!` procedural macro and that should be defined within each project. An example of usage can be found in the Miralis repository[3].

[1] https://crates.io/crates/softcore-asm-rv64
[2] https://crates.io/crates/softcore-rv64
[3] https://github.com/CharlyCst/miralis

References

[1] RISC-V Sail model. https://github.com/riscv/sail-riscv.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019.

[3] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the 26th USENIX Security Symposium*, pages 917–934, 2017.

[4] Charly Castes, François Costa, Nate Foster, Thomas Bourgeat, and Edouard Bugnion. Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet. In *Proceedings of The 20th Workshop on Hot Topics in Operating Systems (HotOS-XX)*, pages 27–33, 2025.

[5] Charly Castes, François Costa, Neelu S. Kalani, Timothy Roscoe, Nate Foster, Thomas Bourgeat, and Edouard Bugnion. The Design and Implementation of a Virtual Firmware Monitor. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2025.

[6] Rémi Delmas, Zyad Hassan, Qinheping Hu, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Adrián Palacios, Celina Val, Michael Tautschnig, Justus Adam, Daniel Schwartz-Narbonne, and Carolyn Zech. Kani: A model checker for rust, 2026.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A Foundry for the Deductive Verification of Rust Programs. pages 90–105, 2022.

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

[9] Ralf Jung. How to use storytelling to fit inline assembly into rust, March 2026.

[10] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.*, 4(POPL):41:1–41:32, 2020.

[11] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.*, 10(POPL):1383–1411, 2026.

[12] Neelu S. Kalani, Charly Castes, Guokai Chen, Alexandre Doukhan, Daniel Bucher, Thomas Bourgeat, and Edouard Bugnion. Flashpoint: Privileged-Software Isolation on RISC-V for Dynamic Root of Trust. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXXII)*, 2027.

[13] 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. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023.

[14] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Alexander Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 234–251, 2017.

[15] Stefan Maus, Michal Moskal, and Wolfram Schulte. Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving. pages 284–298, 2008.

[16] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. ERASan: Efficient Rust Address Sanitizer. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*, pages 4053–4068, 2024.

[17] Magnus Oskar Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, UK, 2009.

[18] 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 (SOSP)*, pages 225–242, 2019.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[20] Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre. coma, an Intermediate Verification Language with Explicit Abstraction Barriers. In *ESOP (2)*, pages 175–201, 2025.

[21] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. Get Rid of Inline Assembly through Verification-Oriented Lifting. pages 577–589, 2019.

[22] Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 2016 Formal Methods in Computer-Aided Design Conferenc (FMCAD)*, pages 161–168, 2016.

[23] Vivien Rindisbacher, Evan Johnson, Nico Lehmann, Tyler Potyondy, Pat Pannuto, Stefan Savage, Deian Stefan, and Ranjit Jhala. TickTock: Verified Isolation in a Production Embedded OS. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, pages 786–801, 2025.

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, 2012.

[25] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying Dynamic Trait Objects in Rust. In *ICSE (SEIP)*, pages 321–330, 2022.

[26] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. Tree Borrows. *Proc. ACM Program. Lang.*, 9(PLDI):1019–1042, 2025.

[27] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA Companion*, pages 301–312, 2011.

[28] Sacha Élie Ayoun, Opale Sjöstedt, and Azalea Raad. Soteria: Efficient Symbolic Execution as a Functional Library. *CoRR*, abs/2511.08729, 2025.