From Rust Till Run: Extending Memory Safety From Rust to Cryptographic Assembly

Shai Caspin
scaspin@princeton.edu
Princeton University
Princeton, New Jersey, USA

Nikhil Pimpalkhare
nikhil.pimpalkhare@princeton.edu
Princeton University
Princeton, New Jersey, USA

Amit Levy
aalevy@princeton.edu
Princeton University
Princeton, New Jersey, USA

Abstract

Memory safety is an important property for security-critical systems, but it cannot be easily extended to cryptography, which is a common source of memory safety vulnerabilities. Cryptography libraries use assembly for direct control of timing and performance, but assembly introduces unsafety when it is called from a high-level memory-safe language like Rust. To enable quick and safe integration of assembly into Rust, specifically for building memory-safe cryptography, we present CLAMS. CLAMS verifies cryptographic assembly against safety constraints derived directly from Rust's type system. Verification is done through symbolic execution at compile time to minimize run-time overheads, and supports verifying loops over potentially unbounded input buffers. CLAMS's procedural macro interface forces developers to map safe Rust types to registers and define preconditions on input and output parameters. CLAMS's techniques can verify the memory safety of assembly from a popular open-source cryptography library. We evaluated CLAMS and found that verification is quick and imposes compile-time overheads under 100ms and negligible run-time overheads.

CCS Concepts: • **Software and its engineering** → **Assembly languages**; **Automated static analysis**; Model checking; • **Security and privacy** → *Cryptography*.

Keywords: Assembly, Automated Verification, Cryptography, Foreign-Function-Interface, Memory-Safety, Rust

ACM Reference Format:
Shai Caspin, Nikhil Pimpalkhare, and Amit Levy. 2025. From Rust Till Run: Extending Memory Safety From Rust to Cryptographic Assembly. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3764860.3768333

1 Introduction

Cryptography underpins the security of modern systems and is essential for secure communication, data protection, and user privacy. To meet the demanding performance requirements and guarantee constant-time execution to avoid timing side channels, cryptographic code is often hand-written in assembly. While assembly improves speed and architectural fit, it makes the code difficult to write, maintain, use, and verify. Subtle bugs, especially highly prevalent memory-safety issues [19, 23, 85], can lead to severe vulnerabilities.

Memory-safe languages like Rust offer strong safety guarantees, but rewriting high-performance cryptographic code entirely in Rust is impractical. Fast Rust cryptography largely relies on unsafe code, such as assembly and nightly (not-yet-stable) SIMD features, or is a wrapper around unsafe C libraries. A single memory-safety bug in foreign code can break Rust's memory safety guarantees. Developers need strong assurances that linked assembly will not compromise Rust's safety, especially in security-critical contexts.

Existing approaches for verifying or integrating foreign assembly do not meet the performance or safety standards necessary for cryptography. Formal verification is accurate but labor-intensive, and is used in practice for select functions or contexts. Developer tools like analyzers and fuzzers help find bugs, but provide no formal guarantees. Isolation techniques at Rust's foreign-function-interface ensure that even unsafe code does not compromise Rust's memory safety, but impose impractical runtime overheads for cryptography and may require hardware or OS support [12, 49, 54, 67, 69].

Integrating assembly into Rust presents additional challenges. Most cryptographic assembly is from mature open-source libraries, commonly reusing assembly from OpenSSL [63] and BoringSSL [42] across libraries, and should not be modified. Bugs often occur when the interface to the assembly is underspecified, so a practical solution should ensure the assembly is called correctly every time.

To bridge the gap between the utility of assembly and the safety of Rust, we present CLAMS: Checking Linked Assembly for Memory Safety. CLAMS is a framework for safely integrating assembly into Rust, providing automatic memory-safety checks of linked cryptographic assembly. CLAMS infers regions for safe memory accesses from safe Rust calling functions, then uses symbolic execution and model checking to verify that the assembly meets memory safety requirements for all possible inputs. Developers can easily map safe Rust types to registers and specify safety preconditions, which are checked at runtime. CLAMS's symbolic execution engine is efficient because it is tailored to support common patterns in cryptographic code: it restricts analysis to functions that access only memory explicitly passed through parameters, it imposes types over registers, it ignores computations that do not influence memory accesses, and it handles loops over potentially unbounded buffers using a loop acceleration technique for constant-step iterations. We used CLAMS to verify 20+ real-world assembly functions from AWS LibCrypto [4] with minimal compile and runtime overhead. For each function, preconditions necessary for memory safety were identified and specified using CLAMS. CLAMS requires no changes to the original assembly, allows seamless Rust integration, and offers developers fast, automatic assurances of memory safety.

2 Memory Safety

Memory safety guarantees the absence of bugs such as out-of-bound accesses, use-after-frees, and double frees. Different programming languages can guarantee memory safety differently, at the cost of added complexity at the language, compiler, and runtime levels. Rust's types, their sizes, bounds checking, ownership, and lifetimes are crucial for Rust's memory safety and must be understood to pass Rust types across the FFI safely. CLAMS extends these concepts down to linked assembly for guaranteed safety when assembly and Rust interact.

2.1 Rust

A large part of Rust's success as an alternative systems language to C is due to Rust's lack of garbage collection and expressive type checking. However, even Rust has a second language, *unsafe Rust*, with support for more programming paradigms outside the limits of memory-safe Rust [6]. Uses of the foreign-function-interface (FFI) require interacting with unsafe Rust, and using the FFI safely requires extensive knowledge of Rust, the foreign language, and the interface itself. Current solutions for isolating foreign code negatively affect performance by separating Rust's and foreign memory at run time [32, 67, 69]. Rust also exposes an unsafe inline assembly macro, asking developers to annotate input and output registers and additional flags, such as whether the function is pure or writes to memory. However, the Rust compiler does not verify these are correct beyond syntax and simply uses them to optimize compilation [76].

Rust achieves memory-safety using techniques including ownership and references, borrowing and lifetimes, typing and mutability rules, and bounds checking. Rust defines a program to be memory safe when all pointer dereferences point to valid, allocated, well-aligned, and accessible memory [31, 73, 75]. Memory safety does not encompass the absence of infinite loops (and thus stack exhaustion) or any termination guarantees.

Rust's ownership model enables tracking that every value has a single owner, and is only valid while its owner is in scope [71]. References are Rust's version of safe pointers and are guaranteed to point to valid memory areas while in scope. Borrowing enables accessing values across scopes and functions. Lifetimes give the Rust compiler more information about how long references live and how they relate, deciding when to free memory. These techniques ensure no memory bugs such as use-after-free and double-frees [74].

All Rust variables are immutable by default or are otherwise annotated using the `mut` keyword, guaranteeing consistency across references and unique access for mutable references [74]. Rust's strict typing is reflected in the knowledge of types, their sizes, and their memory layout.

2.2 Extending Memory Safety to Assembly

Most assembly languages have no safety guarantees and operate over integers or floating-point numbers in registers and memory. Integration of such functions into Rust can be safely achieved by ensuring memory passed to the assembly will not be freed while the assembly is running (ownership), guaranteeing the assembly will not touch memory outside of memory areas explicitly provided by Rust (bounds checks), and ensuring the assembly does not write to non-mutable memory (mutability). To ease the enforcement of these constraints, CLAMS is implemented as a Rust procedural macro to make use of the type checker. CLAMS's implementation as a procedural macro respects Rust's memory model and enforces safety across the FFI, importantly guaranteeing uniqueness for mutable references.

CLAMS checks that for all possible executions, the assembly upholds Rust's requirement that all memory accesses point to valid, allocated, well-aligned, and accessible memory. These enforce non-interference between assembly and Rust, and frame-locality for memory accesses [10]. To check memory safety, CLAMS checks:

1. All memory reads are performed within accessible memory (inputs, stack, or program).
2. All memory writes are performed to accessible and mutable memory (mutable inputs, stack, or program).
3. Restored stack pointer and callee-saved registers.
4. Jumps can only be performed to labels and addresses defined within the same program, i.e., no pointer chasing outside the bounds of the program.
5. No illegal or privileged instructions, including memory allocations and I/O.
6. The program does not depend on any existing external state outside the instructions, the program data, and the inputs and their preconditions.

These rules are checked statically and through symbolic execution, with conditions (1) and (2) requiring a SMT solver.

2.3 Targeting Cryptography

CLAMS targets cryptographic assembly and exploits common patterns such as straightforward control flow and memory access patterns used to maintain constant time guarantees. These functions are deterministic, compute over input in constant-sized blocks, do not allocate or free memory, do not perform I/O, and only use pointers that are provided as parameters or point to labels within the same file. Buffers are passed to the assembly as pointers, often with their length as another variable, and can have variable lengths depending on the algorithm. Other inputs are usually simple and constant-size, such as keys, seeds, or rounds as integers.

These properties simplify reasoning about the memory safety of assembly code, since certain complex behaviors do not have to be accounted for in verification, or can be quickly categorized as unsafe. By targeting verifying cryptography, CLAMS assumes the assembly functions only interact with memory provided using unique pointers, and thus does not have to reason about ownership, lifetimes, or references. CLAMS concludes any function that does not meet the restrictive criteria to be unsafe.

2.4 Related Work

Early work on memory safety introduced Typed Assembly Languages, which imposed types on registers using annotations [30, 59, 60, 79, 79]. The eBPF verifier has similar goals to CLAMS but over the eBPF instruction set [41], and WebAssembly similarly guarantees isolation and memory safety by pointer restrictions and bounds checking [43]. CLAMS enables developers to verify similar safety guarantees without using specialized languages. Other tools enable verifying assembly for concurrency [5, 68, 82, 83], against hardware specifications, and in formal proof engines [39, 48, 68]. These allow developers to verify various properties about assembly, but put the specification and verification burden on the developer, whereas CLAMS does not.

Likewise, related work verifies compilation of cryptography from a high-level specification to assembly [2, 17, 20, 58, 65, 87], or connects a verified compiler to a high-level machine-checkable language [38]. Other libraries formally verify functional correctness of their cryptography using a variety of languages and tools [1, 4, 13, 18, 21, 36]. These solutions are different from CLAMS in that they produce unique assembly code [20, 38], and often require length verification, compilation, and development processes for each algorithm [16, 16, 17, 21, 28].

Symbolic execution and other tools, such as fuzzing and binary analysis, aid in bug finding and safety analysis [9, 14, 22, 23, 33, 34, 61, 64, 84, 84, 85], including heap memory safety [11, 15, 24, 25, 37]. Each is different from CLAMS in that it targets different languages, various security or safety goals, or different techniques for static analysis. KLEE is a mature LLVM symbolic execution tool that has been used to find real-world safety bugs [24], and other tools use symbolic execution to analyze bit-level manipulations of C programs [27, 83, 86]. Prior work also leverages constant-time and loop-based patterns in cryptography for verification [9, 26, 66] similar to CLAMS.

CLAMS's approach for loop invariant generation corresponds to a fragment of affine relation analysis [47, 62]. Related Rust work uses symbolic execution to eliminate panics [55], extract type information for verification from the compiler [7, 8, 44], refine types for stronger preconditions [52], automate reductions to pointer-manipulating programs [56], uncover undefined unsafe behavior [45], or support proof-writing directly [40, 46, 51]. CLAMS can work alongside these since it transforms Rust to Rust, but many of these verification tools cannot model interactions with the FFI. Several efforts aim to isolate calls to unsafe, where bugs are prevalent across multiple interfaces [32, 53, 54, 57, 70], with many proposals for a stronger FFI or isolation using both software and hardware techniques [3, 12, 29, 49, 50, 67, 69, 78, 80, 81] at the cost of overheads.

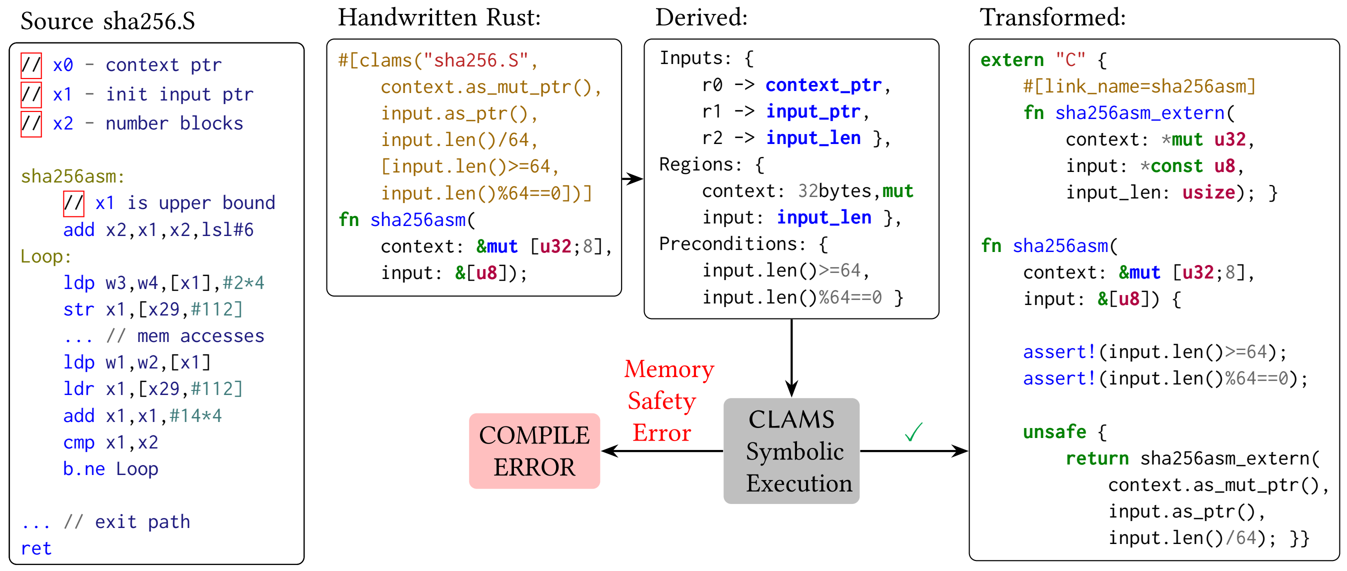
3 Design

CLAMS is a framework for safely calling assembly from Rust, designed to make integration into existing codebases straightforward and practical. CLAMS currently supports verification of assembly that doesn't de/allocate memory or perform I/O, operating over the memory provided in a function's parameters. CLAMS derives safety specifications from Rust function calls and symbolically executes assembly to check the safety of all potential memory accesses.

We implement CLAMS as a procedural macro, which annotates Rust code and transforms it at compile time. CLAMS's flow is shown in Figure 1, where the leftmost code-block highlights instructions pulled from a real SHA-256 function, followed by how a developer would use the CLAMS macro to verify the assembly, all the mappings and preconditions that are derived and used in the symbolic execution, and the final transformed function that safely wraps the assembly call. The produced code includes an unsafe call to the assembly, safely wrapped by a function that checks preconditions. If verification fails, a memory-safety error is thrown at compile time. If the assembly is called with incompatible parameters, either the type checker throws a compile-time error or the assert calls panic at runtime, ensuring the assembly is never run with unsafe parameters.

3.1 Translating Rust Types to Safety Constraints

Instead of assuming memory regions of unknown size that may arbitrarily alias, CLAMS uses information from the Rust type system to define the *shape* of memory the assembly computes over. Naively, when an array is passed to assembly as a pointer, information is lost about the array's elements and size, which may be necessary for performing bounds



Source sha256.S

```
// x0 - context ptr
// x1 - init input ptr
// x2 - number blocks

sha256asm:
    // x1 is upper bound
    add x2,x1,x2,lsl#6
Loop:
    ldp w3,w4,[x1],#2*4
    str x1,[x29,#112]
    ... // mem accesses
    ldp w1,w2,[x1]
    ldr x1,[x29,#112]
    add x1,x1,#14*4
    cmp x1,x2
    b.ne Loop

... // exit path
ret
```

Handwritten Rust:

```
#[clams("sha256.S",
    context.as_mut_ptr(),
    input.as_ptr(),
    input.len()/64,
    [input.len()>=64,
    input.len()%64==0])]
fn sha256asm(
    context: &mut [u32;8],
    input: &[u8]);
```

Derived:

```
Inputs: {
    r0 -> context_ptr,
    r1 -> input_ptr,
    r2 -> input_len },
Regions: {
    context: 32bytes,mut
    input: input_len },
Preconditions: {
    input.len()>=64,
    input.len()%64==0 }
```

Transformed:

```
extern "C" {
    #[link_name=sha256asm]
    fn sha256asm_extern(
        context: *mut u32,
        input: *const u8,
        input_len: usize); }

fn sha256asm(
    context: &mut [u32;8],
    input: &[u8]) {

    assert!(input.len()>=64);
    assert!(input.len()%64==0);

    unsafe {
        return sha256asm_extern(
            context.as_mut_ptr(),
            input.as_ptr(),
            input.len()/64); }}
```

Figure 1. CLAMS derives inputs and memory-safe regions from a Rust function call and macro, used as parameters for symbolic execution. CLAMS developers provide the source assembly and the annotations in the "Handwritten Rust" block, which include parameter remapping and defining preconditions inside the macro. If the assembly is admitted as safe, a wrapper is generated with runtime precondition checks.

checks. Additionally, values across input registers may have hidden yet important relationships; for example, a buffer can be passed as a pointer in one register and a length in another.

Parameter Remapping. CLAMS requires developers to define both the Rust types of parameters and how those values map to registers. As seen in the "Handwritten" block of Figure 1, in the second to fourth lines of the macro, a developer specifies how the safe Rust types in the function declaration (in the last lines of the code block) are passed to assembly. CLAMS requires that all parameters in the function declaration are primitive types so their sizes can be derived during macro expansion. This requirement practically only affects structs, which users should rewrite into a tuple, and enums. Parameters are used in symbolic execution as *symbolics*, for example, `input.as_ptr()` is the symbolic `input_ptr`. Symbolics are unique and indicate that a value is passed as a parameter and could take many values at runtime.

Type	Region?	Methods?	Auto derived?
bool	✕	✕	0 | 1
char	✕	✕	✕
int/float	✕	✕	size, sign
array	✓	as_(mut)_ptr	size, mut
slice	✓	as_(mut)_ptr, len	mut
tuple	✓	as_(mut)_ptr, field	size, mut

Table 1. Mappings of Rust types to information used by CLAMS. A type can represent a region (arrays, slices, and tuples, passed by reference), can be transformed using methods to assembly parameters, and can imply invariants over these parameters, such as region size or mutability.

Table 1 shows which Rust types are translated to memory regions (those passed by reference and larger than registers), which methods can be used to convert Rust types to assembly parameters, and what information can be derived at compile time about each parameter. In types that compose others, such as an array or a tuple, the base element types must also be primitive. CLAMS allows passing slices to assembly if the slice length is another input parameter, as shown in Example 1 with the *input* slice. Structs must be rewritten as tuples to specify subfield types, since macros cannot access any struct declarations. The macro reconstructs the struct in a C representation before passing it via a pointer, to ensure the memory layout is as expected, since Rust may otherwise optimize the representation. Mutability maps to permissions on memory, such that mutable references can be written to, but otherwise a memory region is read-only.

Constructing memory regions. Each safe memory region is distinct and accessible by a single pointer, holds byte-indexed contents, and has a size and a mutability. For the example in Figure 1, there are two regions: the `context` region of 32 bytes and the `input` region with the symbolic length `input_len`. A memory access that includes the symbolic `context_ptr` will access the `context` region. The size may be an integer, such as 32B, or symbolic, such as `input_len`. Regions are treated as logically isolated, i.e., the context pointer cannot be used to access the input buffer.

Defining Preconditions. Assembly is often written with stricter preconditions than can be defined by Rust's type system. For example, the verification of a function may rely on the precondition that an input slice's length is a multiple of some value and non-zero. While some constraints could be expressed using primitive types, CLAMS enables developers to define richer assertions to refine the input types using Rust's syntax for comparison and lazy boolean expressions. The preconditions are type-checked by Rust and evaluated at run-time using asserts, and can be written using any input variable by name. In the running example from Figure 1, they are specified within the square bracket list, which is the last input to the macro. The preconditions are often necessary for verification since they describe what the assembly assumes the inputs will look like. In the case that a precondition is omitted, if the precondition failing could lead to memory safety bugs, CLAMS will fail to verify the assembly. The necessary preconditions for CLAMS to verify a function is safe are a subset of all preconditions. If a precondition is not met, running the assembly could lead to a memory safety error and even a segmentation fault.

3.2 Checking Memory Safety

CLAMS checks memory safety by building a symbolic model of a computer at the start of a program's execution and transforming the model down all possible paths of execution. Since cryptography is often written to be constant-time, the control flow patterns of programs yield a tractable number of branches to explore without merging branches. Loops over potentially infinite input buffers are an exception and require additional techniques to resolve. The rules for memory safety outlined in Subsection 2.2 are checked throughout execution. For example, register restoration is checked at the end of each execution branch, but branching on unset flags will cause an error when reading the flag during the instruction execution. Illegal instructions cause an error when they are reached during symbolic execution. When a memory access is performed, CLAMS generates an SMT query to check if the access is safe and cannot be outside a valid memory region. CLAMS concludes that a program is memory-safe when all branches of execution do not include any unsafe behavior.

Symbolic Execution. CLAMS's symbolic execution is tailored to cryptographic assembly in that it imposes types on registers, only tracks state relevant to memory accesses, and can accelerate and verify constant loops quickly. We divide symbolic execution across two core components: a computer containing symbolic memory (including a stack and program data) and registers, and an engine that handles control flow and sends instructions to the computer one at a time. The engine manages a call stack and adds constraints to an SMT solver to track how execution evolves. The computer is initialized with the memory model from the Rust types, program data, and values in input registers and the stack. Initially, all flag values are set as `unset`, and callee-saved registers are set with their name as a symbolic value like `sp, ra` for special instructions and tracking that they are restored at the end of execution.

A register is typed, and can be an `Address`, `Immediate`, `BaseOffset`, or a `SymbolicInput`. Only address types can be used to access memory. A pointer passed as a parameter will be stored as a symbolic value within an `Address` type register. `SymbolicInput` types are an optimization to discard values that cannot lead to valid memory accesses, for example, an unset value read from an input buffer. This optimization allows CLAMS to ignore changes over input values and only focus on state changes that will affect memory accesses or branch decisions as execution continues. Each instruction reads and transforms the relevant computer state, overwriting registers, memory, and flags based on the operation and data types. A load or store will first check whether an address (stored in a register) is memory safe, and then move values between registers and memory accordingly. CLAMS utilizes a symbolic grammar that mimics Z3's AST using Rust enums to model abstract expressions and comparisons over symbolics in registers and state tracking. Z3 [35] is used for evaluating the safety of a memory access, checking the feasibility of branches, and storing information learned about symbolic values down each execution path.

Evaluating Safety of Memory Accesses. We consider an address (stored in a register) access to be memory safe when:

1. There is one and only one pointer in the address.
2. Writes access mutable memory regions only.
3. The offset is greater than or equal to 0.
4. The sum of the offset and the width of the memory access (how many bytes are accessed) is strictly less than the size of the memory region and well-aligned.

Conditions (1) and (2) and alignment are checked statically. Conditions (3) and (4) are checked using a solver. For example, if s is the offset of the accessed region, w is the width of the access, and l is the length of the region, we verify that a memory access is safe if an SMT solver returns that $s < 0 \lor s + w \geq l$ is unsatisfiable. Conditions (3) and (4) require SMT solvers for more complex scenarios where symbolics show up in the offset and lengths of a memory region. Memory is modeled as multiple regions for inputs, the stack, and program data, and is modeled as byte-addressable and input pointer parameters are assumed to be well-aligned. The stack is treated as a special case with negative offsets, and stack values are only accessible for overflowed inputs or after they are written by the program. Program data is accessed using labels as addresses translated by the engine.

Branching. When an instruction depends on evaluating a comparison, such as for a branch, we consider the boolean expression a *condition*, and whether it evaluates to true or false a *decision*. In the A64 ISA, the instruction pattern is usually a compare operation (such as `cmp`) followed by a branch, for example `bne`. Branch conditions are derived by comparing two register values, or in this case, symbolic expressions, and composing the flags to reach a branch decision.

Using the SMT solver, both decisions are evaluated for feasibility. To track the decisions made along an execution, we use a call stack that includes the state of all registers and flags, the condition, the decision, and all memory accesses performed since the last decision. At a branch, the execution may continue down either decision path or both. In the case where both are feasible paths, the state of the computer is cloned, an appropriate decision frame is pushed onto each decision stack, and symbolic execution continues down each path. CLAMS does not merge execution branches.

Loops over input buffers. The SHA-256 example outlined in Figure 1 loops over blocks in the input buffer and an array in program data that introduces a source of randomness. If a program loops over a buffer using a symbolic length to branch out, execution naively continues infinitely down a branch. An insight is that all loops over input buffers can be described as transforming relevant registers by a *constant step* for every iteration, since the algorithms compute over blocks of input. To prevent infinite execution, CLAMS uses an acceleration technique to resolve constant-step loops. CLAMS computes inductive memory safety proofs by simultaneously proving (1) that there exists a constant difference between symbolic state before and after every iteration of the loop and (2) that all iterations of the loop are memory-safe.

The first two iterations of the loop are used to generate a candidate step difference and rewrite the register state in terms of a looping variable k. CLAMS then simulates the kth iteration of the loop using this step difference as an abstraction, ensuring all memory accesses are still safe for the iteration bounds on k. This shows that for any iteration k, memory accesses within the loop are safe given the entry condition (i.e., the decision made at the end of the previous loop iteration to keep looping). When the inductive proof is completed, the exit condition is checked for feasibility using the solver, the loop is accelerated, and execution continues assuming that the exit condition is met.

To ensure termination, CLAMS concludes unsafety when the depth of the decision tree reaches $2^b \times MAX$, where b is the number of branch instructions in the program and MAX is the size of the largest region. The upper bound ensures that any loop over an input buffer or `.data` memory will be fully explored. The bound on depth is only reached in practice when not all necessary preconditions are specified.

4 Case Study: AWS LibCrypto

We verify cryptographic Aarch64 assembly from Amazon's AWS LibCrypto (aws-lc) [4], which borrows assembly from OpenSSL [63] and BoringSSL [42]. We found that many of the functions had underspecified safety-critical preconditions on the lengths of input buffers.

Engineering overhead. Programmer effort for using CLAMS requires rewriting the foreign function call using the macro and modifying parameters and preconditions. These efforts are highlighted for various functions in Table 2. Most developer effort was spent uncovering preconditions on the lengths of inputs, since they were rarely documented in the actual code, and verification would fail without them, since the SMT solver could not prove that the first memory accesses were within some non-zero bound.

family	# funcs	param remap	struct to tuple	length asserts	complex asserts
SHA, MD5	7	✓	✕	✓	✕
AES	5	✓	✓	✓	✓
GHASH	3	✓	✕	✓	✕
Keccak, SHA-3	3	✓	✕	✓	✓
bignum	2	✓	✕	✓	✓
ChaCha	3	✓	✕	✕	✕

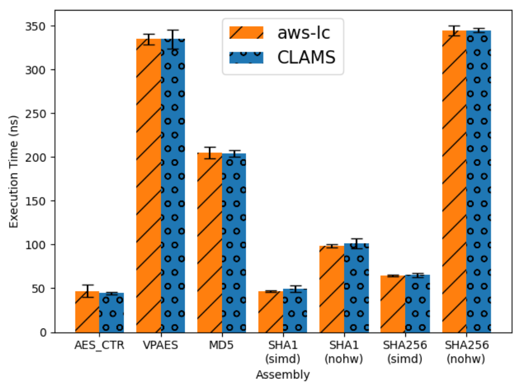
Table 2. Engineering efforts (shown in each column) for successful verification with CLAMS. Parameter remapping includes mapping Rust to assembly types, struct-to-tuple conversions are used when a struct is passed to the assembly and has to be deconstructed, and asserts refine parameter types and their connection.

SHA-256 operates over 64-byte blocks of input and saves the result in the buffer of the mutable context parameter. CLAMS requires assertions on the length of the input buffer. In this case, two assertions are needed: that the input length is a multiple of 64 and greater than 0. All functions labeled as "length asserts" in Table 2 have similar preconditions over their buffer length. For AES, the inputs contain a key schedule data structure with a buffer large enough for all keys and the number of rounds. The number of rounds is used to index into the schedule buffer in the `keys` struct, so it is necessary to define that the number is 10,12, or 14.

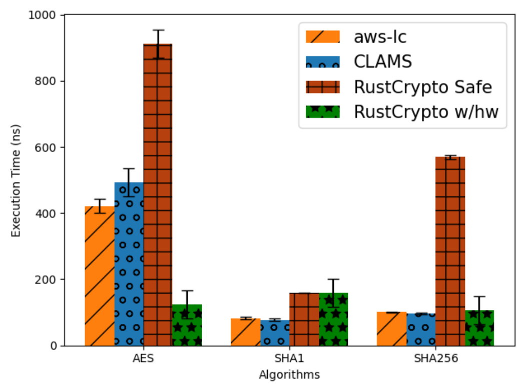
Bignum arithmetic, including addition, subtraction, and multiplication, is implemented in assembly and takes in multiple buffers, but the lengths of all the buffers are equal and are composed of 16-byte blocks. The SHA-3 algorithm operates in two parts: absorb and squeeze. The length of the output of SHA-3 is variable, but at most 1600 bits. In aws-lc, the output buffer is provided as a set size state array of `&[[u64;5];5]`. It also takes in an input as a pointer, the input length, and a variable r for the block size to be processed. The input can be of any length, including 0, and at each invocation, the largest multiple of |r| out of |len| bytes is processed, and the rest is returned. There is a hidden relationship between the bit-width, r, and the capacity that can be described using preconditions, but the output is always written to the 5 by 5 array, so these preconditions do not influence the safety of memory accesses into this array.

4.1 Performance

The constraints derivation and symbolic execution are performed at compile time, while the runtime is affected by additional function wrappers, precondition asserts, and struct rewrites. All of the measurements in this section were collected using Rust's benchmarking framework on an Apple MacBook Pro with an M1 Pro processor running rustc 1.81.0 using the nightly channel.



(a) Run times of only assembly for aws-lc and CLAMS.



(b) Run times of an encrypt function.

Figure 2. Comparing run times of aws-lc (Rust, C, and assembly), CLAMS (Rust and CLAMS assembly), RustCrypto Safe (safe Rust), and RustCrypto with hardware acceleration (unsafe Rust). Comparisons for only running the assembly and full algorithm implementations.

Build and compile times. The CLAMS library and proc macro build in 0.8 and 0.4 seconds each, but with dependencies, they take 14.5s to build, including 7 seconds for Z3 dependencies (not Z3, but Z3 Rust language support). Verification time is on the scale of microseconds, with all assembly routines verified in under 100ms each. The fastest verification time is for bn functions since they are short (around 30 lines), and the longest are for the no-hardware AES and SHA (500+ lines), since they have many instructions and require resolving loops using the specialized acceleration technique. AES and SHA class functions take around 80 ms to verify.

Runtime overheads. We measure the overheads imposed by CLAMS when calling the assembly directly in Figure 2a. As shown, CLAMS imposes little to no overhead at runtime compared to calling the same assembly functions when linking directly against the aws-lc source. On average, the CLAMS runtime of the same function is 0.2% slower, within the standard deviation. The evaluation shows that the overheads imposed by the wrapping functions, struct reconstructions, remappings, and precondition checking are negligible.

Rewriting C glue code. To evaluate whether CLAMS aids in rewriting C-based cryptography libraries and provides a performance advantage over existing unsafe versions, we rewrote some of aws-lc's C code in Rust. We used CLAMS to wrap the calls to assembly, and then rewrote all the C code connecting to the public function calls. The benchmark is the time it takes to generate and encrypt a random input of 16 blocks. In Figure 2b, we compare the rewritten Rust code against the original Rust and C code from aws-lc (accessible using unsafe) and RustCrypto [77] implementations with a safe full-Rust version and an unsafe version with hardware support. As shown in Figure 2b, the safe version of the algorithm in only Rust is inefficient; the hardware-optimized version from RustCrypto is faster but unsafe. Only the CLAMS and all-Rust implementations are safe, showing CLAMS is a step forward in speeding up safe cryptography.

5 Discussion

CLAMS is currently limited to supporting cryptography functions with simple memory access patterns and control flow. We want to extend CLAMS to support other applications that rely on assembly for performance, like video encoding and decoding [72], math libraries, and possibly all assembly use cases in the Rust ecosystem, which would require new definitions of memory safety for interactions with non-input memory. We want to optimize our safe cryptography to achieve comparable speeds to unsafe implementations. We want to extend the CLAMS toolbox to derive semantics from Arm's public ISA specifications, support more ISAs and languages, be compatible with Rust's inline interface and check its flags, and automatically generate preconditions.

6 Conclusion

We introduce CLAMS, a framework for connecting assembly code safely to Rust. CLAMS extracts memory safety constraints from safe calling Rust code and verifies the assembly using symbolic execution and a new acceleration protocol for constant-step loops. CLAMS' interface allows specifying parameter remappings and preconditions. We verified and benchmarked assembly from AWS LibCrypto and found that CLAMS imposes minimal compile and run-time overheads. In particular, CLAMS is good at forcing developers to specify preconditions for verification, setting a promising path for safely integrating assembly into Rust for performance.

Acknowledgments

We thank the reviewers for their valuable feedback and Josh Aas for his insights. This work was funded in part by the National Science Foundation under Grants No. #2303639, #2443589, and a gift from the Internet Security Research Group. Shai Caspin is supported by an NSF GRFP No. #2039656 and a Gordon Wu Fellowship in Engineering.

References

[1] Martin R Albrecht and Kenneth G Paterson. 2016. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*. Springer, 622–643.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1807–1823.

[3] Hussain MJ Almohri and David Evans. 2018. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 248–255.

[4] Amazon Web Services. 2024. *AWS LibCrypto*.

[5] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 303–316.

[6] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[7] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. doi:10.1145/3360573

[8] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (Oct. 2019), 30 pages. doi:10.1145/3360573

[9] Anish Athalye, Henry Corrigan-Gibbs, M Frans Kaashoek, Joseph Tassarotti, and Nickolai Zeldovich. [n. d.]. Modular Verification of Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*.

[10] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C Pierce. 2018. The meaning of memory safety. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. Springer, 79–105.

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[12] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. 2023. TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium (USENIX Security 23). Baltimore, MD: USENIX Association*.

[13] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-aided cryptography. In *2021 IEEE symposium on security and privacy (SP)*. IEEE, 777–795.

[14] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. 2015. Symbolic Execution for {BIOS} Security. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*.

[15] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–68.

[16] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified correctness and security of {OpenSSL}{HMAC}. In *24th USENIX Security Symposium (USENIX Security 15)*. 207–221.

[17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R Lorch, et al. 2017. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*.

[18] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 445–459.

[19] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. 2024. Cryptography in the Wild: An Empirical Analysis of Vulnerabilities in Cryptographic Libraries. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (Singapore, Singapore) *(ASIA CCS '24)*. Association for Computing Machinery, New York, NY, USA, 605–620. doi:10.1145/3634737.3657012

[20] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying {High-Performance} Cryptographic Assembly Code. In *26th USENIX security symposium (USENIX security 17)*. 917–934.

[21] Brett Boston, Samuel Breese, Joey Dodds, Mike Dodds, Brian Huffman, Adam Petcher, and Andrei Stefanescu. 2021. Verified cryptographic code for everybody. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 645–668.

[22] Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. 2019. Understanding how to use static analysis tools for detecting cryptography misuse in software. *IEEE Transactions on Reliability* 68, 4 (2019), 1384–1403.

[23] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A {Static/Symbolic} Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium (USENIX Security 20)*. 199–216.

[24] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[25] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*. 1066–1071.

[26] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 69–76.

[27] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. 2024. Practical Verification of System-Software Components Written in Standard C. (2024).

[28] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. 2018. Continuous formal verification of Amazon s2n. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 430–446.

[29] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[30] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35.

[31] Will Crichton. 2018. Memory Safety in Rust. https://stanford-cs242.github.io/f18/lectures/05-1-rust-memory-safety.html

[32] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[33] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2023. BINSEC/REL: symbolic binary analyzer for security with applications to constant-time and secret-erasure. *ACM Transactions on Privacy and Security* 26, 2 (2023), 1–42.

[34] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 653–656.

[35] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[36] Brett Decker, Benjamin Winters, and Eric Mercer. 2021. Towards verifying SHA256 in OpenSSL with the software analysis workbench. In *NASA Formal Methods Symposium*. Springer, 72–78.

[37] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Holger Hermanns and Jens Palsberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–302.

[38] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Operating Systems Review* 54, 1 (2020), 23–30.

[39] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[40] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. Refinedrust: A type system for high-assurance verification of Rust programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1115–1139.

[41] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1069–1084.

[42] Google. 2024. *BoringSSL*.

[43] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 185–200.

[44] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 426–454.

[45] Ralf Jung. 2020. Understanding and evolving the Rust programming language. (2020).

[46] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[47] Michael Karr. 1976. Affine Relationships among Variables of a Program. *Acta Inf.* 6, 2 (jun 1976), 133–151. doi:10.1007/BF00268497

[48] Andrew Kennedy, Nick Benton, Jonas B Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. 13–24.

[49] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 132–148.

[50] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 51–57.

[51] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 286–315.

[52] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1533–1557.

[53] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.

[54] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2022. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*. Springer, 680–700.

[55] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114.

[56] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (Oct. 2021), 54 pages. doi:10.1145/3462205

[57] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2024. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. *arXiv preprint arXiv:2404.11671* (2024).

[58] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. 2021. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust.* Ph. D. Dissertation. Inria.

[59] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 1998. Stack-based typed assembly language. In *International Workshop on Types in Compilation*. Springer, 28–52.

[60] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

[61] Nicky Mouha and Christopher Celi. 2023. A vulnerability in implementations of SHA-3, SHAKE, EdDSA, and other NIST-approved algorithms. Springer, 3–28.

[62] Markus Müller-Olm and Helmut Seidl. 2004. Precise Interprocedural Analysis through Linear Algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 330–341. doi:10.1145/964001.964029

[63] OpenSSL Foundation. 2024. *OpenSSL*.

[64] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[65] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. 2020. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 983–1002.

[66] Tobias Reinhard, Justus Fasse, and Bart Jacobs. 2023. Completeness Thresholds for Memory Safety of Array Traversing Programs. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 47–54.

[67] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference*. 824–836.

[68] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 825–840.

[69] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. 2025. Building Bridges: Safe Interactions with Foreign Languages through Omniglot. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA: USENIX Association.*

[70] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

[71] Steve Klabnik and Carol Nichols, with contributions from the Rust Community. 2024. *The Rust Programming Language - Understanding Ownership*. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[72] The Rav1d Maintainers. 2024. *Rav1d AV1 Decoder.*

[73] The Rust Contributors. 2023. *The Rust Reference – Unsafety.* https://doc.rust-lang.org/reference/unsafety.html

[74] The Rust Contributors. 2023. *The Rustonomicon.* https://doc.rust-lang.org/nomicon/

[75] The Rust Contributors. 2023. *The Rustonomicon – How Safe and Unsafe Interact.* https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html

[76] The Rust Contributors. 2024. *The Rust Reference - Inline Assembly.* https://doc.rust-lang.org/reference/inline-assembly.html

[77] The RustCrypto Organization. 2024. *RustCrypto.* https://github.com/RustCrypto

[78] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1249–1278.

[79] Hongwei Xi and Robert Harper. 2001. A dependently typed assembly language. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 169–180.

[80] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 649–665. doi:10.1145/3620665.3640408

[81] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (2010), 91–99.

[82] Dachuan Yu and Zhong Shao. 2004. Verification of safety properties for concurrent assembly code. *SIGPLAN Not.* 39, 9 (sep 2004), 175–188. doi:10.1145/1016848.1016875

[83] Dachuan Yu and Zhong Shao. 2004. Verification of safety properties for concurrent assembly code. *ACM SIGPLAN Notices* 39, 9 (2004), 175–188.

[84] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 643–653.

[85] Yuanhang Zhou, Fuchen Ma, Yuanliang Chen, Meng Ren, and Yu Jiang. 2023. CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 45, 28 pages. doi:10.1145/3628160

[86] Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. 2022. BFF: foundational and automated verification of bitfield-manipulating programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 182 (Oct. 2022), 26 pages. doi:10.1145/3563345

[87] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1806.