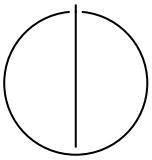
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of SHA-3: Proof Techniques for Symmetric Cryptography

Tristan Schwieren



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

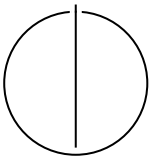
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of SHA-3: Proof Techniques for Symmetric Cryptography

Formale Verifikation von SHA-3: Beweistechniken für symmetrische Kryptographie

Author:	Tristan Schwieren
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Julian Pritzi
Submission Date:	16.11.2025



Acknowledgments

I thank my advisor, Karthikeyan Bhargavan from Cryspen Sarl, for his support and for answering my many questions.

I also thank my supervisor at the Technical University of Munich, Julian Pritzi, for his feedback and review. As well as Dennis Sprokholt for his guidance.

Finally, I thank my friends and family for their support.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 16.11.2025 Tristan Schwieren

Abstract

We formally verify a production-grade Rust implementation of SHA-3 using HAX, a Rust verification toolchain. We address the challenge of verifying loop intensive, bit manipulating cryptographic code with traits and SIMD (AVX2, NEON) paths, and in the process uncover and eliminate a latent edge case panic in the extendable output function buffering API. Our machine checked proofs establish memory safety and panic freedom across portable and vectorized variants with modest proof overhead. This demonstrates that practical, optimized cryptography can be mechanically verified within an existing Rust codebase, substantially strengthening assurance beyond testing and fuzzing.

Zusammenfassung

Wir verifizieren formal eine produktionsreife Rust Implementierung von SHA-3 mit HAX, einer Verifikationstoolchain für Rust. Wir adressieren die Herausforderung, schleifenintensive, bitmanipulierende kryptografische Software mit Traits und SIMD Pfaden (AVX2, NEON) zu verifizieren, und decken dabei einen latenten Randfall auf, der in der Extendable Output Function (XOF) Puffer API zu einer Panik führt, und beheben ihn. Unsere maschinell geprüften Beweise stellen Speichersicherheit und Panikfreiheit für portable und vektorisierte Varianten mit geringem Beweisaufwand sicher. Dies zeigt, dass praktische, optimierte Kryptografie innerhalb einer bestehenden Rust Codebasis mechanisch verifiziert werden kann und die Vertrauenswürdigkeit deutlich über Tests und Fuzzing hinaus erhöht.

Contents

1 Introduction

Cryptography secures every aspect of our digital lives. From online banking to private messaging to every website we visit. Cryptography is the foundation of information security. When cryptography fails, the consequences are tangible: Attackers may be able to read private messages, impersonate users, or manipulate financial transactions. A prominent example is the Heartbleed bug (*The Heartbleed Bug* 2014) in OpenSSL, which was caused by a buffer over-read and allowed attackers to read sensitive data from server memory. Millions of passwords and other secrets were exposed on major web services, including healthcare portals, financial institutions, and government sites.

It is therefore crucial that the cryptographic algorithms we use are secure. Design flaws or implementation bugs can render these algorithms vulnerable. Design flaws in widely deployed cryptographic algorithms are comparatively rare: Candidate schemes are standardized over many years and scrutinized in multiple rounds using both traditional cryptanalysis and modern verification techniques. Implementation bugs, on the other hand, are the more common and pressing problem. They remain frequent and can lead to critical vulnerabilities (Blessing, Specter, and Weitzner 2021).

The common issue is that the most widely used cryptographic libraries are implemented in C or C++. This predominance can be attributed to three factors: Their longstanding heritage, with OpenSSL first released in 1998, high-performance requirements, and the need for portability across diverse architectures, instruction sets, and operating systems. To meet these demands, libraries such as OpenSSL often provide multiple implementation variants that are optimized for different platforms and hardware extensions.

Developers use C and C++ to gain fine-grained control over memory management and system resources. While this benefits performance, it also increases the risk of introducing security vulnerabilities. The US government has recognized this problem and advises against using C or C++ for new software development (National Security Agency, Cybersecurity and Infrastructure Security Agency, and Federal Bureau of Investigation 2022). Instead, it recommends using memory-safe languages such as Java, Python, or Rust.

Cryptographic software developers find Rust especially interesting because it is a modern, general-purpose, multi-paradigm programming language that emphasizes performance and memory safety, while providing developers with fine-grained control over system resources. It achieves memory safety without a garbage collector through its ownership model and borrow checker, while delivering performance comparable to

C and C++. For these reasons, it has gained popularity in systems programming as well as in the development of cryptographic libraries such as RustCrypto (RustCrypto Project 2017) and libcrux (Cryspen 2022).

Rust's borrow checker and type system already prevent many classes of bugs: They rule out use-after-free, double frees, and some buffer overflows in safe Rust. Rust's bound checks at runtime come at a cost, but also prevent some memory safety vulnerabilities. However, developers can still introduce bugs, for example by edge case pointer arithmetic, implementing a cryptographic algorithm wrongly or by accidentally leaking timing information through data-dependent branches. To reason about these higher-level correctness and security properties that go beyond Rust's built-in guarantees, **we use formal methods to prove properties about our code, such as complete memory safety, functional correctness, constant-time execution, and termination.**

In the past, formal methods have been successfully used to build cryptographic libraries. Researchers use the (*Verified Software Toolchain* 2025) (VST) as a framework to verify C programs in the Rocq proof assistant. Appel 2016 formally verified the functional correctness and safety (absence of buffer overruns) of the OpenSSL SHA-256 implementation using VST. The HACL* library (Zinzindohoué, Bhargavan, Protzenko, and Beurdouche 2017) is a formally verified cryptographic library written in F* and automatically translated to C. It provides implementations for many cryptographic primitives. Other proof assistants such as EasyCrypt (*EasyCrypt* 2025), CryptoVerif (Blanchet 2025), and Lean (*Lean Theorem Prover* 2025) have also been used to verify cryptographic algorithms and protocols.

In practice, we first rely on Rust features, testing, and fuzzing to eliminate many common implementation bugs quickly. We want to first catch low-hanging fruit with minimal effort before investing in the more expensive process of formal verification.

Formal verification remains a tedious process that requires expert knowledge in both formal methods and cryptography. This is exacerbated by the fact that distinct verification tasks typically require distinct tools: Researchers use F* to verify memory safety, termination, and functional correctness, while ProVerif is better suited to prove properties about protocols and their security (Protzenko and Raad 2024). As a result, different parts of a single system may need to be reasoned about in different proof assistants, which is cumbersome for developers.

In this thesis, we explore the HAX tool chain (Bhargavan, Buyse, Franceschino, et al. 2025) to formally verify a Rust implementation of the Secure Hash Algorithm 3 (SHA-3) (*FIPS Publication 202: SHA-3 Standard: Permutation-based Hash and Extendable-Output Functions* 2015). HAX promises to make formal verification more accessible and easier to use by connecting Rust code to several proof backends. It can utilize F*, Rocq, Lean, SSProve, and ProVerif as proof backends, so that the same Rust implementation can be analyzed with different tools depending on the verification task, while still benefiting from Rust's strong type system and memory safety guarantees.

We suspect that implementations of round-based, bit-manipulating cryptographic

primitives face unique challenges when being formally verified. Such primitives include hash functions, block ciphers, and stream ciphers. Bit-level operations make it harder for automated provers to reason about them while loops over arbitrary buffers may introduce subtle memory bugs. We demonstrate that formal methods can reveal small edge cases that lead to memory bugs, even in memory-safe implementations, and we assess the strengths and weaknesses of the HAX tool chain on such code.

We make three contributions. First, we prove panic freedom and termination of both portable and SIMD optimized versions (AVX2 and NEON) of SHA-3. Second, we show how formal methods surface memory bugs and pitfalls when iterating over buffers in a supposedly memory-safe Rust implementation: Our verification uncovers and helps fix an edge-case bug in the XOF buffering API that can lead to a panic under unintended but plausible use, demonstrating that formal methods can still add value on top of Rust's safety guarantees. Third, we distill proof-engineering patterns that make verification of such code feasible and that we expect to be reusable for other formally verified, high-performance cryptographic implementations. Additionally, we show that formal methods can simplify function design by encouraging clearer, leaner refactorings.

2 Background

This chapter begins with core concepts in cryptography and its primitives, then focuses on hash functions with an emphasis on SHA-3 and the sponge construction. We next survey implementation libraries and introduce Rust as the systems programming language context for our implementations. Finally, we turn to formal methods, covering proof assistants and the HAX toolchain, that enable rigorous reasoning about cryptographic code.

2.1 Cryptography

Cryptography forms the foundation of modern information security by providing mathematical tools to protect digital communications and data. This field addresses threats from adversaries who may attempt to eavesdrop, modify, or forge digital information. In protocol analysis, a common abstraction is the Dolev-Yao adversary, who controls the network but treats cryptographic primitives as perfect black boxes. In contrast, the standard computational (complexity-theoretic) view models adversaries as probabilistic polynomial-time algorithms and defines security via games and reductions (Katz and Lindell 2007). In this thesis, however, the focus is on the safety of concrete implementations of these primitives, rather than on the design or formal security proofs of the underlying schemes.

Cryptographic systems aim to preserve five fundamental security properties. *Confidentiality* ensures that sensitive information remains accessible only to authorized parties. *Integrity* protects data from unauthorized modification, enabling detection of any tampering. *Availability* ensures that systems and data remain accessible to legitimate users even under attack. *Authenticity* verifies the identity of communicating parties, preventing impersonation. Finally, *non-repudiation* provides proof of message origin and delivery, preventing parties from falsely denying their actions.

2.1.1 Primitives

Several foundational cryptographic primitives serve as building blocks for protocols and systems. These include:

- Symmetric Encryption, such as AES-256 or ChaCha20, is a shared-key primitive used to ensure that only holders of the secret key can read the encrypted data, thus protecting confidentiality.

- Hashing is a one-way transformation that does not use a key and is designed to guarantee data integrity through pre-image and collision resistance. It also underlies authenticity mechanisms in digital signatures.
- Authenticated Encryption with Associated Data (AEAD), for example, AES-GCM or ChaCha20-Poly1305, combines symmetric encryption and integrity checks in a single algorithm, protecting both the secrecy of the plaintext and the integrity of the message, as well as its associated data.
- Key Exchange (e.g. X25519) is an asymmetric protocol that enables two parties to derive a common secret over an insecure channel, ensuring the confidentiality of the resulting shared key.
- Signatures are an asymmetric primitive that allow a signer to prove message authenticity and integrity, and provide non-repudiation by preventing the signer from later denying authorship.

2.1.2 Hash Functions

In this work, we focus on SHA-3 (*FIPS Publication 202: SHA-3 Standard: Permutation-based Hash and Extendable-Output Functions* 2015). Hash functions are particularly interesting because they serve as a foundational building block for many cryptographic systems. Secure implementations of hash functions have different challenges compared to other primitives, such as signature schemes. Where signature schemes rely on big integer arithmetic over secret keys, hash functions primarily operate on bit-level manipulations over multiple rounds. They process arbitrary messages, with buffering and padding logic that turns inputs into fixed-size blocks. Reasoning about such implementations requires proving that the buffer management is correct for all possibilities and that padding and absorption interact as intended, in addition to proving the correctness of the core permutation.

They take an input of arbitrary format and length and always produce a fixed-size output, called the hash or digest. A hash function is said to have *n-bit strength* if any generic attack requires work on the order of 2^n operations (for preimages and second preimages), and on the order of $2^{n/2}$ operations for collisions, due to the birthday bound. Cryptographic hash functions have the following properties:

- *Deterministic*: The same input always produces the same output.
- *Preimage resistance*: Given a target hash value, finding any input that maps to it is infeasible. A hash of n bits has an expected preimage resistance strength of n bits.
- *Second preimage resistance*: Given an input and its hash, finding a different input that yields the same hash is infeasible, with the same n-bit strength.

- *Collision resistance*: Finding any two distinct inputs that produce the same hash is infeasible. Due to the birthday paradox, the expected strength is $n/2$ bits.
- *Avalanche effect*: A small change in the input (even a single bit) should produce a significant change in the output. Every output bit should change with a probability of 50%.

Over the years, numerous cryptographic hash functions have been developed, standardized, and subsequently broken. As surveyed by Preneel (2012), the landscape of cryptographic hash functions has evolved through successive designs, standardization, and cryptanalytic breakthroughs.

MD5, designed by Rivest in 1992, follows the Merkle-Damgård construction. It was later shown to be vulnerable to practical collision attacks, beginning with Wang, Yin, and Yu's result in 2005 (Wang and Yu 2005). Subsequently, the SHA family increased security and output length. SHA-1 produces a 160-bit hash and has since been deprecated after the first collision was found in 2017 (Stevens, Bursztein, Karpman, et al. 2017). SHA-2 offers variants with 224, 256, 384, or 512 bits, all of which are still based on the Merkle-Damgård construction. In response to repeated breaks of Merkle-Damgård based designs, NIST launched the SHA-3 competition to standardize a new primitive with a different design. Keccak was selected and standardized, introducing the sponge construction.

2.1.3 Secure Hash Function 3

Secure Hash Algorithm 3 or SHA-3 emerged from a competition organized by the National Institute of Standards and Technology (NIST) to address vulnerabilities in earlier hash functions based on the Merkle-Damgård construction (Preneel 2012). While the SHA-2 algorithm is still considered secure, NIST sought to create a new standard based on a different construction, distinct from its predecessors, and capable of withstanding attacks from both classical and quantum computers.

SHA-3 includes fixed output lengths of 224, 256, 384, and 512 bits, as well as an extendable output function (XOF), namely SHAKE128 and SHAKE256. Other finalists in the competition were BLAKE, Grøstl, JH, and Skein.

The selected algorithm, Keccak, introduced a novel approach known as the sponge construction. This flexible method allows for the absorption of any amount of data and the generation of a hash of any desired length. During the absorption phase, Keccak processes the input data in fixed-sized blocks and incorporates it into its internal state using the Keccak permutation function. The permutation consists of multiple rounds and is made up of five steps: θ, ρ, π, χ, and ι. Should the last block be smaller than the fixed size, it is padded. The squeezing phase is similar in that it produces an output block and then applies the permutation again. Keccak can squeeze out a partial block if the desired output length is not a multiple of the block size by truncating the final

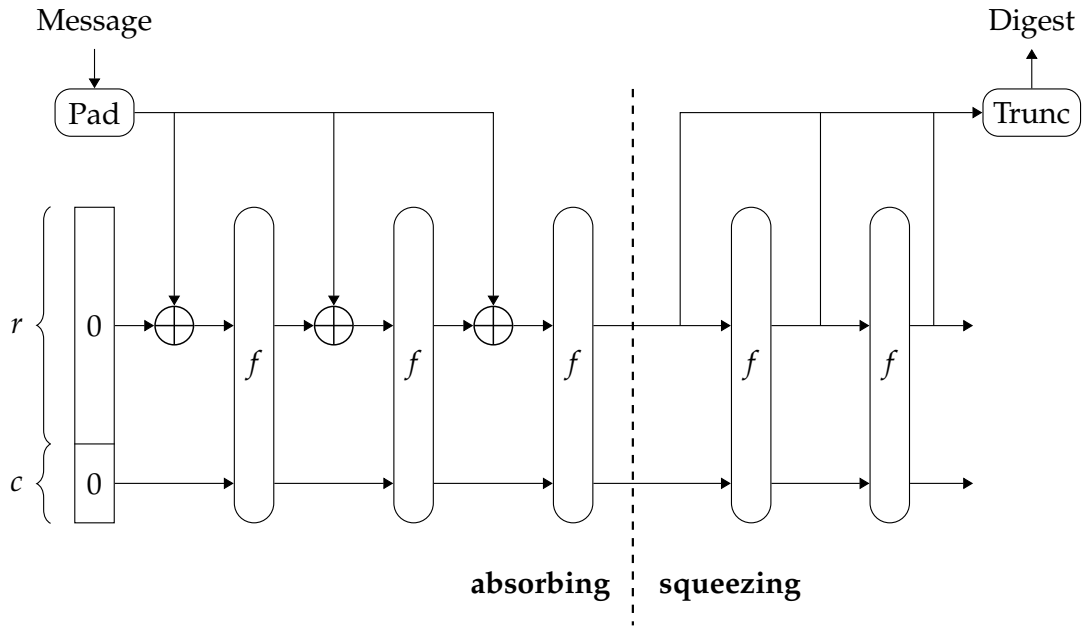


Figure 2.1: Sponge Construction

block. θ mixes columns by xoring column parities into neighboring columns. ρ rotates each lane by a step-dependent offset. π permutes the positions of the lanes. χ applies a non-linear bitwise combination within rows. ι xors a round constant into the state to break symmetry.

Figure 2.1 illustrates the sponge construction, using SHAKE256 as an example. The Keccak permutation f operates on a state divided into rate ($r = 1088$ bits or 136 bytes for SHAKE256) and capacity ($c = 512 = 1600 - 1088$ bits) portions. The absorbing phase XORs padded input blocks into the rate and applies permutation f. The squeezing phase extracts output by repeatedly permuting and reading from the rate, then truncating to the desired length. This example could have a 360-byte message that is padded to 408 bytes ($= 3 * 136$), resulting in a 180-byte digest truncated from 272 bytes ($= 2 * 136$).

2.1.4 Libraries

Cryptographic libraries serve as essential building blocks rather than stand-alone applications, each striking its own balance between algorithm support, performance, and API design. For example, OpenSSL spans hundreds of algorithms across over 670,000 lines of C code, whereas libsodium deliberately narrows its scope, providing a safer and easier-to-audit subset in under 100,000 lines. Libraries that exploit hardware extensions for maximum throughput often incur greater complexity and increased code size compared to more minimal implementations. This complexity also manifests in

Name	Language	Maintainers
OpenSSL	C	OpenSSL Project
LibreSSL (OpenSSL fork)	C	OpenBSD Project
BoringSSL (OpenSSL fork)	C++	Google
WolfSSL	C	WolfSSL Inc.
Crypto++	C++	Community Maintained
libsodium (NaCl fork)	C	Frank Denis and more
GnuTLS	C	GNU Project
MbedTLS	C	ARM Ltd.

Table 2.1: Popular cryptographic libraries

their interfaces: Low-level APIs provide fine-grained control over parameters, albeit at the risk of misuse, whereas high-level, opinionated wrappers reduce developer errors by enforcing safer defaults. Despite these differences, most leading libraries share a common core: They're open source, written in C or C++, and implement fundamental primitives such as SHA-256 and AES-GCM.

TLS is the most widely used protocol that uses multiple cryptographic primitives and is implemented in all but libsodium. Implementing a protocol is much more complex than implementing a single primitive. Protocols must consider several additional aspects, including state management, error handling, message parsing, and network communication. This increases the code's complexity.

Some software projects develop their own cryptographic libraries instead of using an existing one. These typically implement only the algorithms required for their specific use cases. Some examples include GNU Privacy Guard (GnuPG), which uses Libgcrypt, for email encryption and signing. Bitcoin Core, which uses libsecp256k1 for signatures and hashing. The Linux kernel has its own implementation as well.

Different programming languages also have their own cryptographic libraries. Go has the `crypto` package, which includes a wide range of algorithms in the standard library. For Java and C#, there is the Bouncy Castle library. Rust has multiple libraries, the most notable one being RustCrypto. Python, Ruby, JavaScript, and other interpreted languages usually have bindings to C libraries such as OpenSSL.

Table 2.1 summarizes widely used *C* and *C++* cryptographic libraries. These projects are mature, production-grade implementations that are widely deployed in practice, rather than academic prototypes. We restrict the table to low-level, general-purpose libraries, since these are closest in spirit to the implementations we study.

2.1.5 Rust

Rust is a general-purpose systems programming language that focuses on performance, strong static typing, and memory safety without relying on a garbage collector (Klabnik,

Nichols, and Community 2019). Its defining feature is an ownership and borrowing model enforced at compile time by the borrow checker (Matsakis and Klock 2014). Every value has a single owner, and references may be either shared (immutable) or unique (mutable). This model prevents standard classes of memory errors (dangling pointers, double frees, and data races) while preserving zero-cost abstractions that allow Rust to perform on par with C or C++.

A key distinction between Rust and C/C++ lies in how undefined behavior is handled. In C and C++, many erroneous operations (out-of-bounds accesses, use-after-free, dereferencing null pointers, or violating aliasing rules) produce undefined behavior (Kernighan and Ritchie 1988). In safe Rust, these classes of errors and bugs are often caught at compile time or will result in a panic at runtime. The type system, together with the borrow checker, ensures that references are always valid and that mutable aliasing rules are respected. Array and slice indexing in Rust performs bounds checks (unless explicitly removed in unsafe code or optimized away by the compiler). Option and Result types make absent values and fallible operations explicit, and ownership guarantees ensure that resources are released deterministically.

For cryptography, reducing memory-safety bugs is a practical win: Many exploitable vulnerabilities in cryptographic libraries stem from memory bugs (National Security Agency, Cybersecurity and Infrastructure Security Agency, and Federal Bureau of Investigation 2022). Rust's safety guarantees make such bugs far less likely in safe code, improving the baseline assurance of implementations. At the same time, Rust preserves low-level control. It does, however, not magically solve all safety concerns. Constant-time programming to avoid timing side-channels remains a semantic property that requires disciplined coding. Rust programs avoid many memory bugs, but they can still happen and cause the program to panic, i.e., by miscalculating a pointer when accessing a buffer.

2.2 Formal Methods

In computer science, formal methods are defined as "the use of techniques from logic and discrete mathematics in the specification, design, and verification of computer systems, both hardware and software. The fundamental goal of formal methods is to capture requirements, designs, and implementations in a mathematically based model that can be analyzed rigorously" (Butler, Miller, Potts, and Carreno 1998). These techniques provide a foundation for ensuring the correctness and reliability of complex systems through mathematical analysis.

Imagine writing a program where every possible input and execution path could be checked before the program is ever run. This is analogous to having a mathematical proof that your software will behave as intended, regardless of the inputs it receives or the sequence of operations it performs. For instance, consider a simple function that sorts a list of numbers. While testing can demonstrate that the function works for

specific inputs, formal methods enable us to prove that the function will correctly sort any list of numbers, regardless of its size or complexity.

Formal methods are particularly valuable in safety-critical systems, such as aviation software or cryptographic implementations, where errors can have severe consequences (Bhargavan, Buyse, Franceschino, et al. 2025). By using formal methods, engineers can uncover subtle bugs and design flaws that might otherwise go unnoticed until they cause a failure after deployment.

2.2.1 Provable Properties

Cryptographic software is commonly analyzed with respect to two broad classes of properties: (i) Cryptographic properties of primitives and protocols (for hashes, e.g., preimage and collision resistance) and (ii) Code-level properties of concrete implementations. This subsection focuses on code-level properties. The cryptographic properties relevant to hash functions are summarized in Subsection 2.1.2.

Panic freedom (termination without runtime exceptions) is a central code-level property. It states that, for all inputs, execution does not trigger runtime errors and therefore completes normally. In Rust, typical sources of panics include out-of-bounds indexing, division by zero, failed assertions, and attempts to unwrap absent values (`Option::None` or `Result::Err`). Establishing panic freedom rules out these failure modes and, in safe Rust, closely aligns with memory-safety guarantees such as bounds safety and the absence of invalid references.

Functional correctness asserts that the implementation satisfies its specification for all inputs. Formally, this is expressed as a refinement or equivalence between an executable model of the implementation and a high-level specification given by preconditions and postconditions. Specifications typically abstract away low-level concerns such as buffer management and overflow checks to describe the intended input-output behavior. A practical challenge is that many standards present algorithms in prose or pseudo-code rather than in a machine-checked specification language.

2.2.2 F*

F* is a programming language, a proof assistant, and a verification toolchain (Beurdouche 2020). Programs are written in an ML-like language. Specifications and proofs are expressed as types and implementations. Many proof obligations are discharged automatically by an SMT solver (Z3) or interactively via tactics and metaprogramming (Martínez, Ahman, Dumitrescu, et al. 2019).

F* supports dependent types: Types may mention values, enabling precise specifications of relationships between inputs and outputs (for example, a vector type indexed by its length, or a buffer whose size appears in the type). Preconditions and postconditions can be encoded as refinements on types so safety and functional properties often reduce to type-checking. Refinement types (a base type paired with a logical

predicate) provide a lightweight idiom for common invariants. For example, a function computing the sum of a list can have a return type that asserts the result equals the mathematical sum of its input elements.

Proof-oriented programming in F* is practical: Developers write specifications (pre- and postconditions, refinement types, and indexed datatypes) alongside their implementations, and F* generates mathematical proofs that the code meets its specifications. To reduce the manual proof burden, F* employs a combination of automation techniques, including SMT-based proof discharge, symbolic reduction (normalization), and a tactic engine for interactive or domain-specific proof scripts.

F* is not just for pure functional code. It provides an open system of user-defined effects to model imperative features such as mutable state, exceptions, concurrency, and custom program logic. This makes it suitable for verifying low-level, effectual code. For example, the Low* Domain-Specific Language (DSL) targets C-like, stack-and-heap programming and extracts verified C. Vale provides a verified assembly backend for cryptographic kernels. Pulse is a DSL for programming with mutable state and concurrency, with specifications and proofs in Concurrent Separation Logic.

Compilation and integration are pragmatic concerns for F* users. By default, F* code is extracted to OCaml, but fragments can be emitted to F# or compiled to C and WASM. The extracted code typically contains no runtime-proof overhead. Specifications and proofs are removed, leaving efficient, idiomatic code that interoperates with larger systems.

F* is a free software developed by Inria, Microsoft Research, and the wider community.

2.2.3 HAX

HAX is a verification toolchain designed to verify Rust software formally (Bhargavan, Buyse, Franceschino, et al. 2025). It is free software developed mainly by Cryspen Sarl. It addresses the unique challenges of verifying complex systems by supporting multiple proof backends, each of which is useful for different verification tasks. The toolchain has three main components: a frontend, an engine, and backends as depicted in Figure 2.2

The frontend parses and type checks Rust code, producing an abstract syntax tree (AST). The engine then applies multiple transformations to this AST to convert it to the proof backends. These transformations include reorganizing code elements, removing direct variable changes, streamlining control structures, and converting loops into functional forms. After these changes, the system sends the modified code to one of the backends.

HAX supports several backends, including F*, Rocq, SSProve, ProVerif, and experimental support for Lean. Each backend has its own approach to handling Rust constructs and proving properties about the code. For instance, the F* backend gen-

erates a purely functional model of the Rust code and verifies it for panic freedom. Similarly, the Rocq backend utilizes Rocq's interactive proof style, SSProve focuses on computational security proofs, while ProVerif automates security protocol verification.

A key feature of HAX is its support for formal specifications within Rust code. Developers can add pre- and postconditions, invariants, assertions, and lemmas to their code. HAX aims to expedite verification and enable developers to maintain specifications and proofs consistent with the evolving code.

HAX also provides formal models for Rust's core library. For primitive types and functions, HAX uses handwritten models to leverage existing libraries and abstractions in each backend. To ensure the accuracy of its translations and library models, HAX combines testing and verification. HAX keeps the generated models executable and tests them against the original Rust code. Additionally, the system uses property-based testing to create tests from the pre- and postconditions within the code.

Several projects have utilized HAX to verify security and correctness properties, including cryptographic libraries like libcrux, protocol implementations such as Bertie, and smart contracts. The toolchain is designed to be modular and extensible, allowing developers to add new backends and integrate with other verification tools. (Bhargavan, Buyse, Franceschino, et al. 2025)

The goal of HAX is to make formal verification more accessible and require less domain knowledge in formal methods.

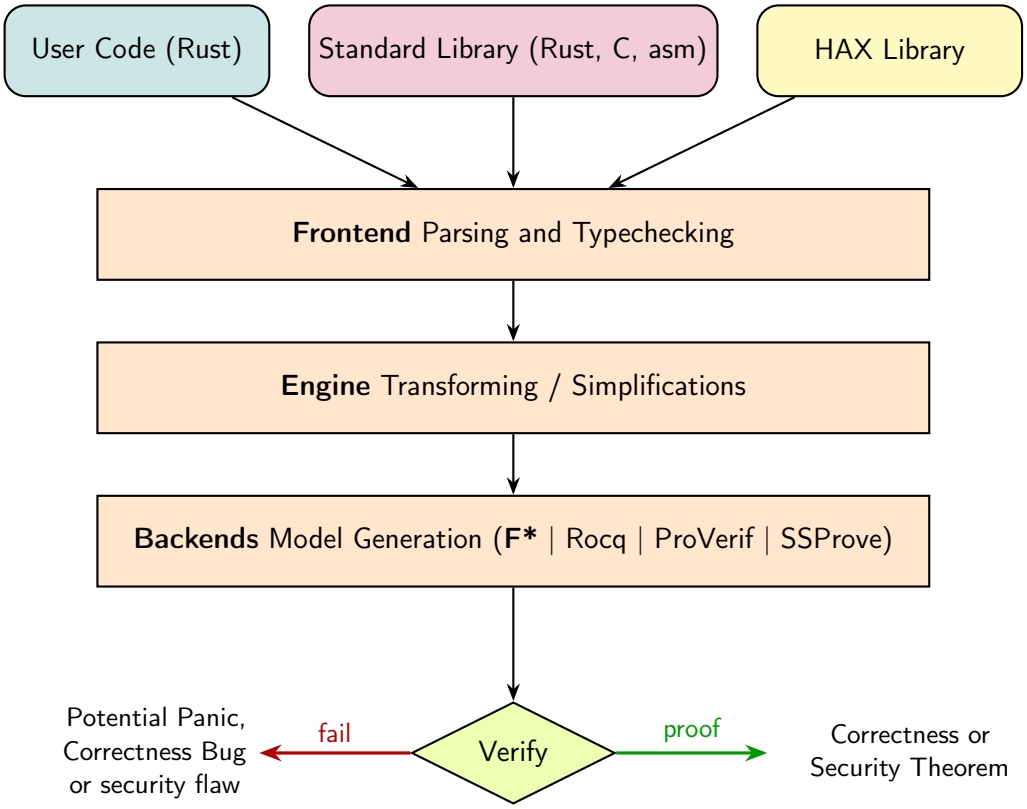


Figure 2.2: Inner workings of HAX

3 Motivation

Cryptography underpins private and secure digital communication, and deployed primitives and protocols must actually deliver their promised security properties (confidentiality, integrity, authenticity, availability, and non-repudiation). Failures arise in two distinct layers: (1) Design flaws in the algorithm or protocol itself and (2) implementation bugs in concrete software. Design assurance is typically high: Standardization bodies such as NIST subject new algorithms to years of scrutiny, and modern proposals increasingly employ formal methods during the design phase. For example, using Tamarin uncovered an impersonation issue in revision ten of TLS 1.3 with 0-RTT resumption workflow (Cremers, Horvat, Scott, and Merwe 2016). Such design-level errors are comparatively rare and are usually identified and mitigated before large-scale exploitation.

Implementation bugs, on the other hand, are much more common. Blessing, Specter, and Weitzner (2021) found using an empirical study of common C/C++ cryptographic libraries (Table 2.1) that "27.2% of vulnerabilities in cryptographic libraries are cryptographic issues while 37.2% of vulnerabilities are memory safety issues, indicating that systems-level bugs are a greater security concern". On average, one new vulnerability appears for every 777-1776 lines of code changed.

The use of memory-safe programming languages, such as Rust, can already prevent many classes of memory bugs that can lead to security vulnerabilities. In fact, this problem has been recognized by the US government which advises against using C or C++ for new software development (National Security Agency, Cybersecurity and Infrastructure Security Agency, and Federal Bureau of Investigation 2022) and instead recommends using memory-safe languages like Rust.

Rust does not magically solve all safety concerns. Rust programs avoid many memory bugs, but they can still happen and cause the program to panic. Over- or underflow of integer arithmetic remains a concern, because it silently wraps around in non-debug builds. **Semantic correctness** is a different problem. It concerns whether the implementation computes exactly the algorithm specified by the standard for all inputs, not just whether it executes safely. This highlights a key point: **Memory safety is necessary but not sufficient**.

For this reason, formal methods for verifying Rust code have gained traction in recent years. One such toolchain is HAX (Bhargavan, Buyse, Franceschino, et al. 2025), which allows developers to write formal specifications and proofs for their Rust code. HAX is used to develop libcrux (Cryspen 2022), a formally verified cryptographic library

written in Rust.

Cryptographic libraries are used by billions of users who perform numerous cryptographic operations daily, such as every time they visit a website using HTTPS or send a private message using an end-to-end encrypted messaging app. Therefore, these libraries must be secure and free of bugs. We argue that the cost of fixing bugs after deployment is much higher than the upfront cost of using formal methods, because failures in the field require incident response, patch development, distribution to all users, and potential downtime, and they impose reputational and compliance costs, whereas proofs prevent classes of defects before release and their cost is paid once and reused.

Formal methods let us state and machine check properties of our code, such as correctness, memory safety, and termination, against a precise specification. Rust's type system and borrow checker, together with testing and fuzzing, establish confidence by sampling behaviors, but only formal methods provide guarantees by ruling out entire classes of bugs for all inputs under explicit assumptions. **We aim for guarantees, not confidence.**

Using formal methods still requires a significant amount of engineering hours and expert knowledge. Therefore, it is not widely used in practice and is only used for high-assurance software. Projects such as HAX aim to make formal verification more accessible, to increase adoption not only among experts in formal methods but also among regular software developers.

This work focuses on the Secure Hash Algorithm 3 (SHA-3). We suspect automatic reasoning about loop intensive round structures is hard, and the large permutations and bit shuffles inflate verification conditions, which makes automated proving harder. Concretely, implementations of round based cryptographic primitives, such as hash functions and block ciphers with extensive bit level manipulation, often need much more guidance for the prover, for example precise loop invariants, staged lemmas, and careful management of the search space. **Bit-level, round-based code is where verification hurts.**

In fact, even the inventors of Keccak, the underlying algorithm of SHA-3, had a vulnerability in the reference SHA-3 implementation. A buffer overflow vulnerability allowed an attacker to write arbitrary data into the system's memory (Mouha and Celi 2023), demonstrating that even experts can make mistakes. This concrete incident supports our suspicion that round based, bit level implementations pose unique challenges for formal verification, because loop intensive round structures are hard to reason about automatically, and large permutations inflate verification conditions, which makes automated proving harder.

We identified a memory bug caused by the unintended use of a lower buffering API, resulting from a pointer miscalculation, which would have caused the libcrux library to panic. This finding reinforces the importance of formal verification in this setting. **Formal verification found what tests missed.** Therefore, this thesis proves

panic freedom and memory safety for both the portable and Single Instruction Multiple Data optimized versions of SHA-3, and assesses the strengths and weaknesses of the HAX toolchain used with F*, to make clear both the guarantees and the engineering trade offs.

4 Design & Implementation

This chapter introduces the libcrux-sha3 crate of the Libcrux cryptographic library, explains our verification approach with HAX and F* (including loops, traits, state invariants, and large state manipulation), outlines the methodology we used in practice, and concludes with the results on memory safety and panic freedom for all SHA-3 backends. We begin with a brief overview of the interactive verification workflow we follow with HAX and F* (see Figure 4.1).

In practice, verification happens in F*: We extract the F* module using HAX, prove interactively, and, once verified, elevate the proof code back to Rust. We typically work in an interactive environment such as Visual Studio Code with the F* extension or Emacs with the F* mode.

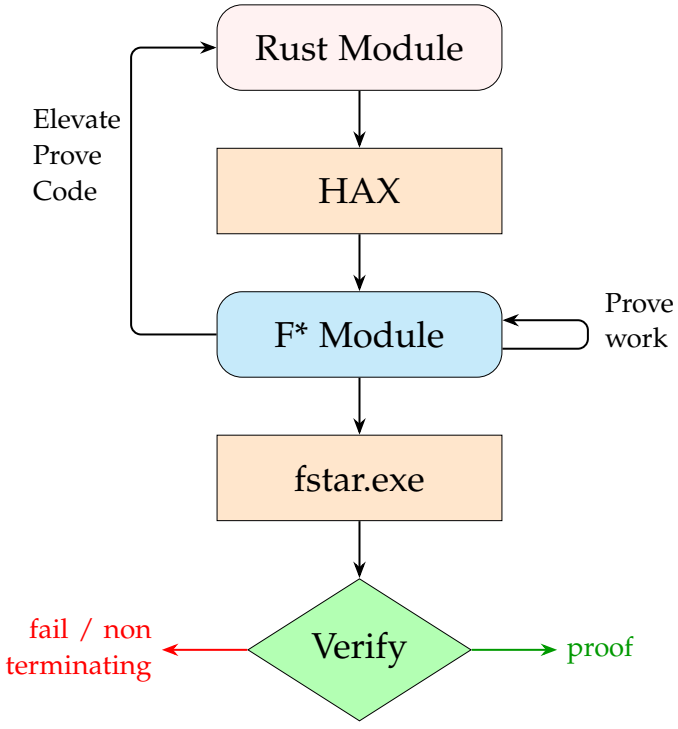


Figure 4.1: Workflow when verifying a Rust Module with HAX and F*

4.1 Libcrux-sha3

Libcrux is a formally verified cryptography library written in Rust by Cryspen. It combines artifacts from HACL*, Fiat Crypto, Vale, Jasmin, as well as its own implementations. All artifacts are accessible behind a Rust application interface, ensuring the correct usage and preserving the verified guarantees.

libcrux-sha3 is one of libcrux's own implementations, which can be used directly or serves as a dependency of other libcrux crates such as libcrux-ml-kem. It implements all versions of the NIST's standardized Secure Hash Algorithm 3. The implementation has three backends: One is portable and does not rely on platform-specific instructions. The two other versions are for `x86_64 AVX2` and `ARM NEON`, and can compute four and two hashes in parallel, given inputs of the same length.

We prove memory safety and panic freedom for libcrux-sha3.

4.2 Verification

4.2.1 Basics of HAX and F*

Before diving into SHA-3 specifics, we outline the general workflow (Figure 4.1) we use with HAX: (1) Write and test ordinary Rust code. (2) Add HAX contracts (`#[hax::requires]`, `#[hax::ensures]`) annotating preconditions and postconditions. (3) Translate the annotated Rust to F*, where contracts become verification obligations (panic freedom, memory safety) to discharge. (4) Work *interactively* in F*: Instead of a one-shot perfect proof, iteratively refine specifications and lemmas in response to F* error messages until all obligations are discharged. (5) Propagate the verified conditions back to the Rust source as HAX annotations.

Contracts are interpreted in the Hoare style: {P} C {Q} means that if P holds before executing C, then Q holds after termination, where `requires` gives P, `ensures` gives Q, and *C* is the function body.

HAX models mutable Rust code as pure F* functions that return updated values instead of mutating in place, and arrays/slices as immutable sequences. The generated F* sometimes looks structurally different from the original Rust, but it encodes the same behavior at the level of memory safety and panics.

F* extends simple types with refinements $x : T\{P\,x\}$, which we use throughout to express bounded integers and sized arrays. HAX systematically turns Rust bounds checks into such refinement types in F*, so that many safety properties become typing obligations. For instance, indices are modeled as refined `usize` values that must lie within the length of the underlying sequence.

We will now begin with an example of a simplified Rust function used in the SHA-3 implementation. Our goal in this example is to show concretely how HAX can be used to verify memory safety and panic freedom for ordinary Rust code. Function `set_ij` in

Listing 1 sets a word in the SHA-3 state at position (x, y), and we use it to illustrate how Rust contracts are translated to F* and discharged as proof obligations.

```
fn set_ij(
  arr: &mut [u64; 25],
  x: usize,
  y: usize,
  value: u64,
) {
  arr[5 * y + x] = value;
}
```

Listing 1: Function `set_ij` for changing the SHA-3 state

The function is simple. However, it has a potential bug. If the caller provides $x \geq 5$ and $y \geq 5$, the function will panic due to an out-of-bounds array access. Thanks to Rust's safety guarantees, we do not have undefined behavior, which is already a significant advantage over C/C++.

The next step is to introduce a debug assertion to check the code during development `debug_assert!(x < 5 && y < 5)`. To ensure the function never panics, we promote this assertion to a precondition, as shown in Listing 2.[1]

F* checks these contracts via weakest preconditions, $\mathrm{wp}(C, Q)$. This is the least predicate on initial states that guarantees Q. In our setting, `#[hax::requires(...)]` is P, `#[hax::ensures(...)]` is Q, and C is the function body.

The HAX toolchain converts this Rust code to F* as shown in Listing 3. We now walk through the generated F*. Lines 1–5 define the function's signature. It takes an array of type `t_Array u64 (mk_usize 25)`, which is HAX's fixed-size array of length 25 with elements of type `u64`. `t_Array` is essentially a wrapper around sequences (`Seq`), which is a built-in F* type. Its definition can be seen in Listing 4. `u64` is HAX's representation of a 64-bit unsigned integer and is a wrapper around F*'s number type with constraints on minimum and maximum values, specifically $0 \leq x < 2^{64}$.

The same applies to the parameters `x` and `y` of type `usize`, which is HAX's representation of Rust's `usize` type. The last parameter is `value` of type `u64`. The function returns an array of the same type as the input array.

This difference also illustrates HAX's F* extraction model. The function in F* has a return type while the Rust function does not because, in Rust, we use a mutable reference to the array, whereas in F* the function is pure and cannot mutate in place. We therefore return the modified array explicitly.

[1] The attribute `#[cfg(not(hax))]` is used here to exclude the debug assertion when compiling for the HAX compilation target (i.e., when generating code for the HAX toolchain). This choice is made solely for the sake of clarity in the example.

```rust
#[hax::requires(x < 5 && y < 5)]
fn set_ij(
  arr: &mut [u64; 25],
  x: usize,
  y: usize,
  value: u64,
) {
  #[cfg(not(hax))]
  debug_assert!(x < 5 && y < 5);

  arr[5 * y + x] = value;
}
```

Listing 2: Function `set_ij` for changing the SHA-3 state with HAX pre-conditions

```
let set_ij
  (arr: t_Array u64 (mk_usize 25))
  (x y: usize)
  (value: u64)
  : t_Array u64 (mk_usize 25)
  = let arr:t_Array u64 (mk_usize 25)
    = update_at_usize arr ((mk_usize 5 *! y) +! x) value
  in
  arr
```

Listing 3: Function `set_ij` in F* generated by HAX

In this example, the index argument of `update_at_usize` is one of these refined `usize` values, so bounds checking becomes a type-level obligation. The type of the index parameter is a dependent refinement:

```
i: usize { v i < Seq.length s }
```

In the same way that a caller of our `set_ij` function has to ensure our preconditions, we have to guarantee the preconditions of the `update_at_usize` function. The `update_at_usize` is also part of HAX and represents the `arr[i] = x` assignment. The precondition of this function is implicit in its type signature at line 8 of Listing 4. The value of `i` has to be less than the length of the Sequence. Other than that, the function is a wrapper around F*'s built-in `Seq.upd` function, which updates a sequence at a given index.

```
type t_Slice t = s: Seq.seq t { Seq.length s <= max_usize }

type t_Array t (l:usize) = s: Seq.seq t { Seq.length s == v l }

let update_at_usize
  (#t: Type0)
  (s: t_Slice t)
  (i: usize {v i < Seq.length s})
  (x: t)
  : t_Array t (length s)
  = Seq.upd #t s (v i) x
```

Listing 4: HAX representation of slices, arrays and array assignment

Looking back at the precondition we added in Rust: $x < 5 \wedge y < 5$, we can see that this is exactly what we need to ensure the precondition of `update_at_usize`. The state array has a length of 25, the maximum values for `x` and `y` are 4, which results in a maximum index of $5 * 4 + 4 = 24 < 25$.

This illustrates how we prove panic freedom for our Rust code using HAX and F*. In practice, **we often discover the right preconditions during verification**, then push them back to Rust as contracts.

4.2.2 Loops

We reason about loops via invariants: A loop invariant is a predicate that holds before the loop, is preserved by each iteration, and helps imply the function's postcondition upon exit. Termination is justified with a well-founded `decreases` measure that strictly decreases on every step. In F*, this is required for recursive proofs and can be used implicitly for loops encoded via recursion.

In our case, loop invariants are primarily used to ensure that a field within a struct remains unchanged during loop execution or that the length of a slice remains constant. When manipulating other fields of a struct or the content of a slice, F* needs help to reason about the entire object.

Loops in SHA-3 iterate over buffers, so we must guide F* in reasoning about how each index is formed. In the simple helper `set_ij` (Listing 3) we discharge the bounds check exactly once. By contrast, a `for` loop introduces a fresh index each iteration, and we have to justify (to F*) that this index remains in range every time.

Formally verifying loops in F* is the most challenging part of proving panic freedom for our SHA-3 implementation. This also distinguishes primitives like hash functions and block ciphers from signature schemes, which typically contain no loops in their

```rust
impl KeccakState {
  #[hax::requires(start + RATE <= blocks.len())]
  fn absorb_block<const RATE: usize>(
    &mut self,
    blocks: &[u8],
    start: usize,
  ) { /* omitted for clarity */ }
}
```

Listing 5: Pre-condition for the `absorb_block` function of the `KeccakState` struct

core algorithms. Looking at the main loop of the Keccak sponge function in Listing 11, we first outline the overall structure and introduce the constant generic parameter `RATE`. In the Keccak sponge (Subsection 2.1.3) the 1600-bit state splits into a rate portion r and a capacity portion c with $r + c = 1600$. Absorption XORs at most r bits of each block before a permutation. Squeezing reads at most r bits after each permutation. We expose r as a compile-time parameter `RATE` = $r/8$. Function preconditions enforce these bounds and required alignment and propagate `RATE` into helper calls. The stated postcondition simply ensures the output slice length remains unchanged.

Keccak works like a sponge: The function first initializes the Keccak state, then absorbs the input data in chunks of size `RATE`. If there is any remaining data, it absorbs the final block with padding. After that, it squeezes the output from the state in chunks of size `RATE` and handles any remaining bytes at the end.

The `absorb_block` function of the `KeccakState` (Listing 5) struct has a pre-condition `start` + `RATE` $\leq$ `blocks.len()`, ensuring that we do not read out of bounds of the input. Other preconditions are omitted for brevity.

Rust for loops are non-inclusive on the upper bound, and integer division rounds down. We iterate from 0 to `input_len/RATE` and then multiply the index by `RATE`, so it is straightforward to see that the precondition holds. The prover, however, needs more guidance. We therefore introduce the arithmetic lemma in Equation (4.1).

$$\forall\, k, n, d \in \mathbf{N}.\; k < n \;\Rightarrow\; k \cdot d + d \leq n \cdot d \tag{4.1}$$

F* can discharge this as simple arithmetic, without reasoning about loops or arrays. This is the minimal help needed to verify the precondition of `absorb_block`. **The lemma was discovered during the verification process**, which typically begins with several local assertions and candidate lemmas. We discuss the proving process in more detail in Section 4.3.

Under Curry-Howard, propositions are types and proofs are (total) programs. In F*, a lemma is a total, ghost function whose body carries the inductive or algebraic reasoning. We call such lemmas in a ghost context (e.g., via `hax::fstar!(...)`), which injects

```
impl KeccakState {
  #[hax::requires(start + len <= out.len())]
  #[hax::ensures(|_| future(out).len() == out.len())]
  fn squeeze<const RATE: usize>(
    &self,
    out: &mut [u8],
    start: usize,
    len: usize
  ) { /* omitted for clarity */ }
}

```

Listing 6: Pre and Post-condition of the `squeeze` function of the `KeccakState` struct

proof-only code that is erased at extraction and has no runtime cost while guiding the SMT about arithmetic facts like `mul_succ_le` (Eq. (4.1)). Concretely, ghost functions (and ghost arguments/returns) are never compiled to runtime code, cannot observe or mutate runtime state, and may only be called in ghost contexts, serving purely as proof artifacts.

The squeezing loop is similar to the absorbing loop. It avoids an unnecessary Keccak permutation when the output length is small or a multiple of `RATE`. It uses the same lemma (Eq. (4.1)) and additionally maintains a loop invariant. This is because the `output` variable is mutable and manipulated by the `squeeze` function. For this reason, `squeeze` has a postcondition (Listing 6) that the length of the output slice remains unchanged. The loop invariant states that this property holds before and after each iteration. F* can then infer the postcondition of `keccak` automatically.

We refer to the post-state value of a mutable variable using `future(x)` in `ensures` clauses. The loop invariant preserves `out.len()` across iterations, so the `squeeze` postcondition composes to the `keccak` postcondition without additional proof burden.

4.2.3 Traits

Traits are a powerful feature in Rust. Traits define shared behavior as a set of required methods, associated types, and constants that types implement. They enable polymorphism via generic trait bounds and can provide default method implementations. Our SHA-3 implementation features three distinct backends tailored to different CPU architectures. One portable implementation, one for x86-64 with AVX2, and one for ARM with NEON. Using traits allows us to define a standard interface for these backends. The portable version hashes a single buffer, while the AVX2 can hash four buffers of the same length simultaneously, and the NEON can hash two buffers simultaneously,

producing either four or two hashes.

As seen before in Listing 11, the main Keccak sponge function uses the `absorb_block`, `absorb_final`, and `squeeze` functions. These functions are defined in the `Absorb` and `Squeeze` traits, which provide a common interface for the different SHA-3 backends.

Implementations must be substitutable for their traits: **The implementation's precondition must be no stronger than the trait's, and the implementation's postcondition must be at least as strong as the trait's.** Trait `requires` implies impl `requires` and impl `ensures` implies trait `ensures`.

In Listing 7, we can see the definition of the `Absorb` trait and its two required methods: `absorb_block` and `absorb_last`. We will be focusing on `absorb_block` as the procedure for `absorb_last` is similar. The trait is generic over a constant parameter `N`, which represents the number of input buffers to absorb. We have the same precondition for RATE as before, but we must also ensure that `N` is non-zero and that all input buffers have the same length. For this, we use HAX's support for propositions. The `forall` statement is equivalent to:

$$\forall i < N.\ \text{len}(input[i]) = \text{len}(input[0])$$

When we implement the trait, we have to ensure that the precondition is equivalent to the one defined in the trait. In Listing 7, we can see the implementation of the `Absorb` trait for the NEON backend. One common mistake is to start verification of the implementation without ensuring that the precondition is equivalent.

4.2.4 State Invariants

Our implementation supports incremental absorption, which can absorb multiple slices of arbitrary length with separate calls before producing an arbitrary-length output. This functionality is part of the Extendable Output Function (XOF), which is used for SHAKE128 and SHAKE256. To support this, we must maintain an internal buffer that holds any data not yet absorbed into the Keccak state. For example, we might start by absorbing 100 bytes, which is less than the rate of SHAKE128 (136 bytes); therefore, we can't absorb and must store. For this reason, we will also only need a buffer of size `RATE`. We reuse the same buffer for subsequent calls and maintain a variable to track the number of bytes currently stored.

All of this is implementation details of the `ShakeXofState` struct and hidden from the user. In fact, the `state`, `buffer`, and `buffer_length` fields are private. The user only has access to the public methods `new`, `absorb`, `absorb_final`, and `squeeze`. Therefore, we can also not use the fields in the pre- and post-conditions of these methods.

Instead, we define a state invariant for the struct. **A state invariant is a property that holds for the internal state of an object throughout its lifetime.** In our case, we want to ensure that the `buffer_length` is always less than or equal to `RATE`. Otherwise, we would read or write out of bounds of the buffer when absorbing. This invariant must

```rust
trait Absorb<const N: usize> {
  #[hax::requires(
    from_bool(
      N != 0 && RATE <= 200 && RATE % 8 == 0 &&
      start + RATE <= input[0].len()
    ).and(forall(|i: usize| implies(
      i < N,
      input[0].len() == input[i].len()
    )))
  )]
  fn absorb_block<const RATE: usize>(
    &mut self,
    input: &[&[u8]; N],
    start: usize
  );

  /* absorb_last omitted for brevity */
}

impl Absorb for KeccakState<2> {
  #[hax::requires(
    RATE <= 200 && RATE % 8 == 0 &&
    start + RATE <= input[0].len() &&
    input[0].len() == input[1].len()
  )]
  fn absorb_block<const RATE: usize>(
    &mut self,
    input: &[&[u8]; 2],
    start: usize
  ) { /* implementation omitted for brevity */ }

  /* absorb_last omitted for brevity */
}
```

Listing 7: Definition of the `Absorb` trait and its implementation for the NEON backend

```rust
fn xof_state_inv(xof: Shake256Xof) -> bool {
    xof.buffer_length <= 136
}

impl Xof for Shake256Xof {
  #[hax_lib::ensures(|result| xof_state_inv(&result))]
  fn new() -> Self { /* omitted for brevity */ }

  #[hax_lib::requires(xof_state_inv(&self))]
  #[hax_lib::ensures(|_| xof_state_inv(&future(self)))]
  fn absorb(&mut self, input: &[u8]) { /* omitted for brevity */}

  /* absorb_final and squeeze omitted for brevity */
}
```

Listing 8: State invariant for the `Shake256Xof` struct

hold after the construction using `new` and is used as a pre- and postcondition for the other methods.

The predicate `xof_state_inv` is the buffer-length invariant over private fields: It is defined as `buffer_length` $\leq$ `RATE` $= 136$ (for `Shake256`). It must hold after construction and be preserved by every public method. We encode this by having the constructor *establish* the invariant (via `ensures`), and every mutator *require* it as a precondition and re-establish it for the post-state `future(self)`. This pattern ensures that public transitions never leave the object in an invalid state.

4.2.5 Large State Manipulation

The SHA-3 version of Keccak is based on an internal state of 1600 bits, or 200 bytes, represented as 25 words of 64 bits each.

When many element-wise writes target a variable, a single long function with repeated assignments can inflate the verification condition and make F* slow or non-terminating. In practice, we avoid this by taking a snapshot `let old = *state;` and creating several small helper functions (e.g., `pi_0`, `pi_1`, ...), each performing just a few writes from the snapshot. **Keeping helpers small reduces the verification condition size, and makes type checking possible while requiring only minor refactoring**. Reducing the size of the verification condition is a common technique in formal verification, in this case we needed to change the source code, which we generally try to avoid. It is a trade-off between source fidelity versus verification.

This changes however, have no runtime cost. Adding the attribute `#[inline(always)]`

```rust
#[inline(always)]
fn pi_0(state: &mut [u64; 25], old: &[u64; 25]) {
  state[1] = old[15]; state[2] = old[5]; state[3] = old[20];
  state[4] = old[10];
}

#[inline(always)]
fn pi_1(state: &mut [u64; 25], old: &[u64; 25]) {
  state[5] = old[6]; state[6] = old[21]; state[7] = old[11];
  state[8] = old[1]; state[9] = old[16];
}

/* pi_1, pi_2, pi_3, pi_4 omitted for brevity */

fn pi(state: &mut [u64; 25]) {
  let old = *state;
  pi_0(state, &old);
  pi_1(state, &old);
  /* pi_2, pi_3, pi_4 calls omitted for brevity */
}
```

Listing 9: Keccakf1600 π permutation

to these helpers is a hint that preserves the original single-function performance by encouraging the compiler to inline the helpers, removing call overhead. Compile time hints, do not affect the Rust-to-F* verification benefit of splitting the source.

In Section 5.1 we show the `Keccakf1600` ρ permutation as an example and the exponential growth in verification size for the monolithic version.

4.3 Dealing with Verification Challenges

This section provides key strategies that we discovered for our SHA-3 verification effort. They include the use of assertions, decreasing the verification condition size by decomposition and exclusion as well as finding needed lemmas and using F* flags.

It targets readers who want to verify their own Rust code using HAX and F*. We would have liked to have been aware of these strategies from the beginning.

When F* fails to verify, it typically displays an error message similar to the one shown below (Listing 10). An assertion failed for some reason. No further information is provided, and we must investigate.

```
error: Assertion failed
SMT solver report:
  unknown because (incomplete quantifiers) (rlimit=20; fuel=0; ifuel=1)

Notes:
  - 'canceled' or 'resource limits reached' means the SMT query timed
    out; consider increasing the rlimit.
  - 'incomplete quantifiers' means Z3 could not prove the query;
    try providing a more detailed proof, or increase fuel/ifuel.
  - 'unknown' means Z3 provided no further reason for the proof failing.
```

Listing 10: Generic F* failure message

4.3.1 Asserting

Using assertions is an effective way to identify the cause of a failed proof in F*. Simple Assertions can be used to validate the inputs or preconditions of functions. Adding additional assertions throughout the function body to validate the intermediate results helps with pinpointing the exact location where the proof breaks.

While assertions are helpful, it is essential to avoid overusing them. Too many assertions can increase the size of the verification condition, potentially causing timeouts or slowing down the verification process. Once an assertion has helped identify or resolve an issue, consider removing it to keep the proof efficient.

Use assertions strategically as a debugging tool, and remove them once they have served their purpose. Generally, only non-trivial assertions in the final version are needed for verification.

4.3.2 Excluding Non-Essential Code

When working on a function, such as the Keccak sponge function in Listing 11, it can be helpful to comment out parts of the code to reduce the size of the verification condition. This of course destroys the functional correctness but can identify memory issues. In that example of the Keccak sponge function, we first removed the squeezing part and the final absorbing. This way, it is more likely that F* will produce usable error messages.

4.3.3 Decomposing Functions

When verification struggles with a large function, decompose it into smaller, focused helpers. Move complex loop bodies or long sequences of element-wise updates into separate functions that operate on a snapshot of the state. Verify each helper independently (small VCs are faster and produce clearer errors), then recompose the main function.

Mark tiny, verification-friendly helpers with `#[inline(always)]` so the compiler can inline them and preserve the original performance.

This is another approach to reducing the size of the verification condition and obtaining more meaningful error messages. This is especially useful when a function contains a loop with a large body. By moving the loop body into a separate function, we can first verify the loop body, identifying any necessary lemmas or invariants. Once the loop body is verified, we can then focus on the loop structure itself, which is usually much simpler.

4.3.4 Finding Needed Lemmas

During the verification process, it may happen that we cannot verify a function with asserts, invariants, and preconditions alone. In our case, we needed the lemma `mul_succ_le` to verify the primary Keccak sponge function (Listing 11) and another lemma about modulo arithmetic for the XOF API. A pragmatic way to discover such missing facts is to temporarily insert `assume` statements in F* that state the property you believe is needed and check whether the proof then goes through. These assumes are strictly a diagnostic aid and must be removed once they have revealed the right statement. Replace each with a lemma declaration and proof. Leaving an `assume` in the final code undermines soundness, whereas the corresponding proved lemma is safe and, in our experience, usually straightforward to establish.

4.3.5 F* Flags

F* provides several command-line flags that can also be used in an interactive environment using the `#push-options "<options>"` directive. These flags can aid in debugging and enhancing verification performance.

The flag `--query_stats` provides detailed statistics about SMT queries and often hints at why a proof failed. The option `--split_queries always` splits SMT queries into several smaller goals, which helps localize errors and results in clearer messages. Instead of the entire function failing to verify, a smaller block may fail. When verifying larger functions, `--z3rlimit` helps by setting the Z3 per-query resource limit and prevents Z3 from aborting a verification it can complete. For example, our high-level Keccak function (Listing 11) requires a higher limit to verify at all. In our experience, the options `--fuel` and `--ifuel` rarely help in the context of verification with HAX, either the proof works with the default values, or it never does, because every function has explicit pre and postconditions. They add value mainly for deep recursive unfoldings, which we generally avoid.

4.4 Results

We proved memory safety and panic freedom for libcrux-sha3 across the portable, AVX2 and NEON backends. All public absorb/final/pad, permutation, squeeze, and incremental XOF functions carry pre- and postconditions whose F* translations discharge bounds, range, and state-shape obligations, and trait implementations (`Absorb`, `Squeeze`) are substitutable for their interfaces. Loop proofs use a single arithmetic lemma (Eq. 4.1) with small invariants, and large state permutations are verified after splitting monolithic updates into inlined helpers.

The proof relied on two, simple lemmas, guided by early bounds failures, targeted loop invariants, and modest decomposition of state updates. Ghost code is erased, so there is no runtime cost. Specifications mirror existing assumptions (valid rates, equal lane lengths) plus a concise state invariant for incremental modes.

```rust
#[hax::fstar::before(r#"
let rec mul_succ_le (k n d: nat) : Lemma
  (requires k < n) (ensures k * d + d <= n * d) (decreases n)
  = if n = 0 then ()
  else if k = n - 1 then ()
  else mul_succ_le k (n - 1) d
"#)]
#[hax::requires(
  RATE != 0 && RATE <= 200 && RATE % 8 == 0 &&
  (RATE % 32 == 8 || RATE % 32 == 16)
)]
#[hax::ensures(|_|
  future(output).len() == output.len())
]
fn keccak<const RATE: usize, const DELIM: u8>(
  input: &[u8],
  output: &mut [u8]
) {
  // Initialize Keccak state
  let mut s = KeccakState::new();

  // Absorb input
  let input_len = input.len();
  let input_blocks = input_len / RATE;
  let input_rem = input_len % RATE;
  for i in 0..input_blocks {
    hax::fstar!("mul_succ_le (v i) (v input_blocks) (v v_RATE)");
    s.absorb_block::<RATE>(&input, i * RATE);
  }
  s.absorb_final::<RATE, DELIM>(
    &input, input_len - input_rem, input_rem
  );

  // Squeeze output
  let output_len = output.len();
  let output_blocks = output_len / RATE;
  let output_rem = output_len % RATE;
  if output_blocks == 0 {
    s.squeeze::<RATE>(output, 0, output_len);
  } else {
    s.squeeze::<RATE>(output, 0, RATE);
    for i in 1..output_blocks {
      hax::loop_invariant!(|_: usize| output.len() == output_len);
      hax::fstar!("mul_succ_le (v i) (v output_blocks) (v v_RATE)");
      s.keccakf1600();
      s.squeeze::<RATE>(output, i * RATE, RATE);
    }
    if output_rem != 0 {
      s.keccakf1600();
      s.squeeze::<RATE>(output, output_len - output_rem, output_rem);
    }
  }
}
```

Listing 11: Main loop of the Keccak sponge function

5 Evaluation

This chapter gives a short overview before the detailed sections. We look at: (1) Performance when verifying large permutations and how helper functions prevent blow ups (2) Limits of HAX types for mutable borrows and why they block a generic multi lane sponge (3) A buffer overflow bug in the XOF code found by proofs and the one line fix (4) Proof size versus implementation size for SHA-3 code (5) How formal methods simplify function design with the `sigma` and `to_u32s` examples of SHA-2

5.1 Verification of Large Permutations

When attempting to type-check an unmodified SHA-3 (ρ) or (π) function, the check never terminates and instead requires more and more resources before the process is terminated or the system crashes. **A naive, monolithic permutation makes F* verification condition blow up.**

The problem is **exponential performance degradation** in the F* type checker that occurs when a function contains many successive let-bindings or update-style assignments. In the observed benchmark Figure 5.1, the time required for type checking and SMT-related (lax) phases grows from fractions of a second to hundreds of seconds as the number of repeated let-bound expressions increases. The root symptom is **formula inflation** in the verification conditions F* generates. When let-bound terms are expanded in line, repeated occurrences of the same subterms are duplicated throughout the verification condition. This duplication substantially increases the cost of normalization and term-manipulation and produces much larger, redundant SMT queries that the backend solver (Z3) must process.

We type-checked prefixes $k = 1..24$ of the real ρ permutation, each variant adding one more assignment from the production code, measuring median clock time (3 runs) on a modern AMD laptop with default F* and Z3 settings. The code is real except for truncation.

The SHA-3 ρ and π permutation shows why this pattern is problematic in code (Listing 9). The permutation is implemented as a sequence of 24 element-wise assignments derived from a single snapshot of the state, followed by many writes from old to the state. Each assignment references structurally related expressions and prior updates. If the verification condition generator expands these expressions in line, every new assignment re-embeds previously expanded structure, producing deeply nested subterms across the generated verification conditions. Because SMT solvers

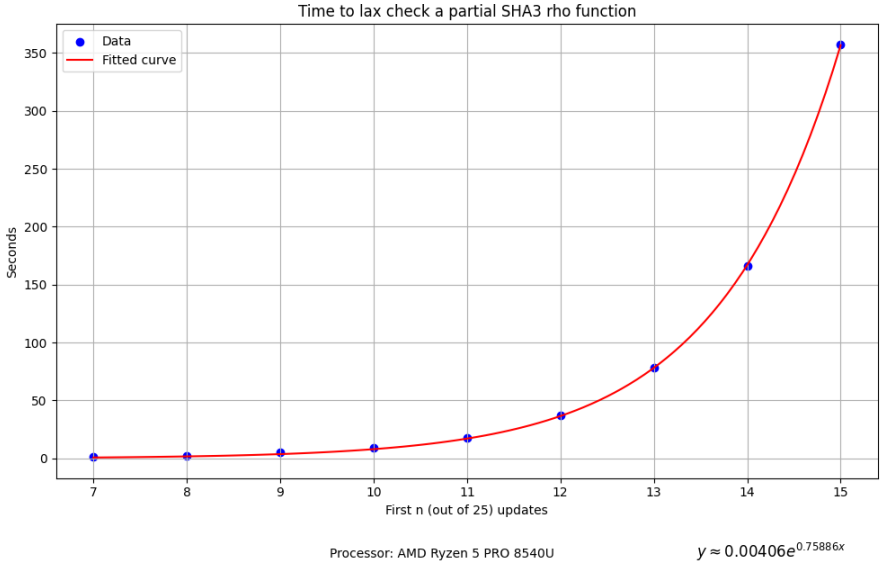


Figure 5.1: Exponential blow up when verifying a partial ρ function

are sensitive to formula size and redundancy, these repeated verification conditions result in significant increases in solver work and memory usage, as well as slowdowns in overall verification time. **Small helper functions over a snapshot fix this without hurting runtime**, because we instruct the compiler to inline them.

5.2 Type Limitation of HAX

Libcrux-sha3 embraces parallel hashing to align with contemporary processor architectures. Modern CPUs expose wide vector units and perform best when several independent streams are advanced in lockstep. We therefore structure hashing as multiple "lanes" that absorb and squeeze concurrently, achieving higher throughput without entangling application-level data flows. Conceptually, the design aims to be parametric in the number of lanes: One unifying abstraction should express single-lane operation as well as multi-lane execution, while preserving the familiar sponge rhythm of writing output and interleaving the requisite permutations.

A central constraint of the verification toolchain complicates this picture. **HAX currently cannot reason reliably about mutable borrows within user-defined types**, nor about aggregates of mutable byte slices. Most notably, arrays of N mutable outputs of the form `[&mut [u8]; N]`. This limitation prevents a generic design. In sponge constructions, squeezing naturally alternates between emitting bytes into mutable buffers and advancing the internal state via a permutation. A generic interface would accept N non-aliasing, length-constrained outputs, and mutate the state in place as it iterates across blocks. However, the verifier cannot discharge the aliasing, exclusivity,

and shape constraints that such an interface requires at a generic boundary. The result is an impedance mismatch between the algorithm's natural formulation and what the verifier can soundly model.

This mismatch forces an architectural compromise. Instead of a single parametric N-lane abstraction, the code must expose several specific variants (e.g., for one, two, or four lanes), each repeating the same high-level logic. The immediate cost is growth in the code surface: Common concerns such as bounds checks, block scheduling, and documentation must be replicated and kept in lockstep across variants, increasing the likelihood of drift. The verification burden scales similarly. Properties that would ideally be proven once for an abstract N must be restated and re-proven for each variant: Non-aliasing and equality constraints among outputs, loop invariants for block iteration, and postconditions about buffer lengths and state progression. This amplifies proof effort, enlarges the search space for the solver, and introduces brittleness when specifications evolve. In short, the present structure is shaped more by the verifier's current modeling limitations than by the cryptographic design. It secures machine-checked guarantees today at the expense of duplication and maintenance overhead, while leaving open the prospect of consolidation should the toolchain's expressiveness improve.

5.3 Bug in the XOF Module

During the formal verification of our SHA-3 implementation, **we discovered a bug** in the generic Keccak XOF (eXtended Output Function) module. **We classify this bug as a typical buffer overflow caused by a miscalculation of indexes**, which would cause the program to panic. Using the internal XOF API in an unintentional but logical way causes the calculated digest to be incorrect. The bug can not be triggered in the intended usage of the SHA-3 library, but only by using the lower-level functionality.

The XOF module provides an interface that allows the user to repeatedly absorb arbitrary-length inputs before squeezing a digest of arbitrary length. We maintain an internal buffer that stores input that has not yet been absorbed into the Keccak state. When a succinct absorb call is made, we either continue filling the buffer or, if the buffer is full, we absorb the buffer into the state. Finally, we continue absorbing whole blocks from the input and store any remaining data into the buffer again.

We first have multiple calls to `absorb` with parts of the input. The amount of data absorbed in each call is arbitrary. Finally, we call `absorb_final` with the remaining input data and the domain separation suffix `0x1f`. After all input has been absorbed, we call `squeeze` to get the digest (Listing 12).

The lower-level XOF API also exposes more fine-grained functions such as `fill_buffer`, which manages combining data from an input slice with the internal buffer. It first checks whether the internal buffer already contains data and, if so, whether, together with the input, there are enough bytes to form a complete block. If so,

```rust
#[test]
fn intended_xof_usage() {
  let mut state = Shake256Xof::new();
  let mut out = [0u8; 32];

  state.absorb(&INPUT[..1]);
  state.absorb(&INPUT[1..136]);
  state.absorb(&INPUT[136..420]);
  state.absorb_final::<0x1fu8>(&INPUT[420..]);
  state.squeeze(&mut out);

  assert_eq!(&out, &DIGEST); // Ok
}
```

Listing 12: Intended usage of the XOF module

it calculates the number of bytes required from the input to fill the internal buffer and copies those bytes from the input into the internal buffer. If the internal buffer is empty and the input alone contains at least one whole block, no data is copied into the internal buffer (the caller can proceed to absorb whole blocks directly). The function returns the number of bytes taken from the input, or zero if the combined data is insufficient to complete a block. This behavior ensures the internal buffer is only updated when a whole block can be formed, simplifying the absorption logic and avoiding partial-buffer writes.

The `fill_buffer` functionality on its own is correct and can be used as intended. It might be used in a scenario where the user wants to fill the internal buffer (which is cheap) without absorbing any data into the state (which is expensive). The functionality might also be helpful when the user is unsure whether the digest will be squeezed later, wants to avoid unnecessary state updates, but still consumes the input.

The edge case where the bug is triggered is when the internal buffer already contains data and the buffer is then filled exactly to capacity using `fill_buffer`. Listing 14 shows this scenario. The next absorb call will then incorrectly not absorb the internal buffer into the state, because `absorb` calls `fill_buffer` again, which will return zero since the internal buffer is already full. After absorbing full blocks, which already miscalculate the state and would lead to an incorrect digest, the remaining input is buffered again, resulting in a buffer overflow because the internal buffer is already at its capacity. The program will then panic. **This is a realistic edge case that testing may miss.**

To understand how formal methods helped us find this bug, we have to look more into how `absorb` works internally:

```
let consumed = self.fill_buffer(input);

if consumed > 0 {
if self.buffer_length == RATE {
  self.state.absorb_block::<RATE>(&self.buffer, 0);
  self.state.keccakf1600();
  self.buffer_length = 0;
}
```

Listing 13: Fixed part of `absorb` function

1. If there is data in the internal buffer (`buffer_length > 0`) and the input has enough bytes to fill the buffer: Fill the buffer from the input and set `buffer_length = RATE` (call to `fill_buffer`).
2. If some input was consumed to fill the buffer: Absorb the full internal buffer into the Keccak state and set `buffer_length = 0`.
3. Compute the number of whole blocks in the remaining input.
4. For each full block: Absorb the block directly into the Keccak state.
5. Buffer any remaining input bytes into the internal buffer and update `buffer_length` accordingly.

In section Subsection 4.2.4, we explain how we use state invariants in XOF, especially that the internal `buffer_length` variable must always be less than or equal to `RATE`. The internal buffer has a size of `RATE`. This invariant could not be proven with the bug present, because in the edge case, the internal buffer would not be absorbed, leaving `buffer_length = RATE` and would continue filling the buffer further using the remaining input and set `buffer_length = RATE + remainder`. This violates the invariant.

The fix is straightforward (Listing 13): In step 2, instead of checking if some input was consumed, check if the internal buffer is full (`buffer_length == RATE`) before absorbing it into the state. **A one-line change restores the invariant and prevents the panic.**

These small edge cases are difficult to detect through testing and are precisely the kinds of faults that formal methods can reveal. Unlike finite input testing, formal verification can establish properties such as panic-freedom for all possible executions. Similar to the reference SHA-3 implementation (Mouha and Celi 2023), and in the absence of Rust's bounds checking, this bug could have allowed an attacker to overwrite stack memory instead of a panic.

```rust
fn unexpected_xof_usage() {
  let mut state = Shake256Xof::new();
  let mut out = [0u8; 32];

  state.absorb(&INPUT[0..32]);
  state.fill_buffer(&INPUT[32..136]);
  state.absorb(&INPUT[136..420]);
  state.absorb_final::<0x1fu8>(&INPUT[420..]);
  state.squeeze(&mut out);

  assert_eq!(&out, &DIGEST); // Fails
}
```

Listing 14: Unintended usage of the XOF module triggering the bug

5.4 Proof Size and Comparison

When comparing the amount of proof work needed in a formal verification project, one metric is the ratio between program code and proof code. This ratio indicates the complexity of the proof effort required. We want to note that this metric, on its own, does not provide an accurate indication of the overall effort, as counting lines of code is not a standardized method. It is easy to write very dense code with few lines but high complexity, or the opposite, with many lines of code but low complexity. The metric serves as a rough estimate and should be primarily used when comparing either the same codebase over time or different codebases in the same language and verification framework with similar code styles.

For the portable and AVX2-optimized SHA-3 implementations, the proof overhead is about **1:3.3 (proof lines : implementation lines)**. In other words, for every line of proof code, there are roughly 3.3 lines of implementation code.

5.5 Formal Methods simplify Function Design

Working with formal methods to prove correctness can help simplify the implementation. When working towards a correctness proof for SHA-2 using an existing implementation as a base, we found that using formal methods helped streamline the implementation, as it made the verification process easier.

5.5.1 Simplifying the `sigma` Function

The first example is the `sigma` function in SHA-2 which in FIPS 180-4 at section 4.1.2 is defined as $\Sigma_0^{\{256\}}, \Sigma_1^{\{256\}}, \sigma_0^{\{256\}}, \sigma_1^{\{256\}}$ that each take only one parameter, the input word x. We define the functions as close as possible to the textual specification in F* as `f_Sigma_0`, `f_Sigma_1`, `f_sigma_0`, `f_sigma_1`.

The original implementation of `sigma` (Listing 15) has several aspects that make it more challenging to verify:

1. Has an additional parameter `op` that selects the upper or lower case sigma function.
2. Uses a mutable variable `tmp`, which is more difficult for F* to reason about.
3. Casts the values of the rotation and shift amounts from `u8` into `u32`, which makes them more compact but with negligible advantage.
4. Uses an array lookup to get the rotation and shift amount, which is a reasonable design decision.

```
1  const OP_TBL: [u8; 12] =
2    [2, 13, 22, 6, 11, 25, 7, 18, 3, 17, 19, 10];
3
4  #[hax::requires(i < 4)]
5  fn sigma(x: u32, i: usize, op: usize) -> u32 {
6    let mut tmp: u32 = x.rotate_right(OP_TABLE[3 * i + 2].into());
7    if op == 0 {
8      tmp = x >> OP_TBL[3 * i + 2]
9    }
10   let rot_val_1 = OP_TBL[3 * i].into();
11   let rot_val_2 = OP_TBL[3 * i + 1].into();
12   x.rotate_right(rot_val_1) ^ x.rotate_right(rot_val_2) ^ tmp
13 }
```

Listing 15: Original Implementation of `sigma`

We can simplify the function by removing the `op` parameter, as it entirely depends on the parameter `i`. We can also remove the mutable variable `tmp` by using a `let` binding. We keep the array lookup for the rotation and shift amounts as it makes the code more compact and easier to read, but we change the type of the array to `u32` to avoid unnecessary casting. While the initial implementation is valid and works correctly, the simplified version (Listing 18) can be verified almost instantly. The extracted F* code is

approximately half the size and does not require the use of tactics because it does not involve type casting. It also proves equivalence to the FIPS specification directly in the function contract.

5.5.2 Simplifying the `to_u32s` Function

Another example is the `to_u32s` function that converts a byte array into an array of 32-bit words. The original implementation (Listing 16) uses a vector to store the output words, which requires heap allocation and is variable in length. The simplified version (Listing 17) uses a fixed-length array that can be allocated on the stack and has a known size at compile time. This change simplifies the verification process because we can now use fixed-size arrays in the function contract and do not have to reason about dynamic memory allocation. The simplified version is also more efficient because it avoids heap allocation. These are all side effects of working with formal verification. Trying to make a function easier to verify often leads to a better overall implementation.

```
fn to_u32s(block: &[u8; 64]) -> Vec<u32> {
  let mut out = Vec::with_capacity(16);
  for block_chunk in block.chunks_exact(4) {
    let block_chunk_array = u32::from_be_bytes(
      block_chunk.try_into().unwrap()
    );
    out.push(block_chunk_array);
  }
  out
}
```

Listing 16: Original Implementation of `to_u32s`

```
fn to_u32s(block: &[u8; 64]) -> [u32; 16] {
  core::array::from_fn(|i| {
    let chunk = block[4 * i..][..4].try_into().unwrap();
    u32::from_be_bytes(chunk)
  })
}
```

Listing 17: Simplified Implementation of `to_u32s`

```rust
const OP_TBL: [u32; 12] =
  [2, 13, 22, 6, 11, 25, 7, 18, 3, 17, 19, 10];

#[hax_lib::requires(i < 4)]
#[hax_lib::ensures(|result| fstar!("
$result =. (match v $i with
  | 0 -> f_Sigma_0
  | 1 -> f_Sigma_1
  | 2 -> f_sigma_0
  | 3 -> f_sigma_1) $x
"))]
fn sigma(x: u32, i: usize) -> u32 {
  let tmp = if i >= 2 {
    x >> OP_TABLE[3 * i + 2]
  } else {
    x.rotate_right(OP_TBL[3 * i + 2])
  };
  let rot_val_1 = OP_TBL[3 * i];
  let rot_val_2 = OP_TBL[3 * i + 1];
  x.rotate_right(rot_val_1) ^ x.rotate_right(rot_val_2) ^ tmp
}
```

Listing 18: Simplified Implementation of `sigma` with Function Contract

6 Related Work

This chapter surveys prior work on the formal verification of cryptographic hash implementations. The related efforts span different verification ecosystems and target languages: (i) F*/Low* with HACL* (verified C extracted from F*), (ii) Jasmin with EasyCrypt (assembly-like code with cryptographic proofs), (iii) Cryptol with SAW (equivalence checking of existing C implementations against declarative specifications), and (iv) the Verified Software Toolchain (VST) with Rocq for C (machine-checked proofs in separation logic). Across these systems, typical proof goals include functional correctness, memory safety, side-channel resistance. Some also provide end-to-end assurance through verified compilation. The following sections summarize representative SHA-3 (and SHA-2) results in each line of work and situate them with respect to this thesis.

6.1 SHA-3 and HACL*

HACL* (Zinzindohoué, Bhargavan, Protzenko, and Beurdouche 2017) is a verified cryptographic library written in Low*. Low* is a subset of F* for low-level programming. Low* code can be translated to C. It allows developers to write memory-safe, low-level code that interacts with hardware. HACL* implements many cryptographic primitives: ChaCha20, Poly1305, AES, Ed25519, X25519, SHA-2, and SHA-3. It also implements TLS. Together with CompCert, a verified C compiler, HACL* gives verification from specification to binary. Properties proven about the F* specification also hold for the compiled program.

The SHA-3 code in HACL* is verified for functional correctness, memory safety, and side-channel resistance. The specification is written in F*. The implementation is written in Low*. Their equivalence is proven in F*. The code is also proven memory-safe. This rules out buffer overflows and use-after-free.

HACL*'s SHA-3 is similar to the work in this thesis. Both use F* for formal verification to leverage its powerful proof automation capabilities using the Z3 SMT solver. The difference is that HACL* uses a domain-specific language that compiles to C. We verify Rust code directly, which enables us to leverage Rust's strong type system and ownership model, catching many memory-safety issues earlier and simplifying verification. HACL* also proves functional correctness. That is out of scope here, but it may be a possibility for future work. Using the ComptCert compiler also allows HACL* to give end-to-end verification from specification to binary, which is not possible with Rust.

6.2 SHA-3 and EasyCrypt/Jasmin

Jasmin is a programming language designed for high-assurance, high-speed cryptographic implementations. Jasmin is assembly-like, eliminating the need for intermediate compilation. This makes Jasmin particularly suitable for performance-critical applications. EasyCrypt, a proof assistant tailored for cryptographic proofs, is used to verify the correctness and security of Jasmin implementations. Together, Jasmin and EasyCrypt provide a robust framework for developing cryptographic software that is both efficient and formally verified.

The SHA-3 implementation in Jasmin (Almeida, Baritel-Ruet, Barbosa, et al. 2019) achieves four critical properties simultaneously: Functional correctness, provable security, side-channel resistance, and efficiency. The implementation is proven functionally correct by demonstrating equivalence to the SHA-3 specification, as written in EasyCrypt. Side-channel resistance is ensured by proving that the implementation does not leak information through timing side-channels. The execution time only depends on the length of the input. Notably, the Jasmin implementation matches the performance of OpenSSL.

Compared to our work, Jasmin's direct compilation to assembly reduces the trusted computing base and preserves performance. The formal proofs in EasyCrypt provide stronger security guarantees, including indifferentiability and side-channel resistance. However, writing assembly-like code in Jasmin is a lot more error-prone than writing in Rust. Our implementation is independent of specific hardware and also optimized for performance using SIMD instructions.

6.3 SHA-3 and Cryptol/SAW

Cryptol is a domain-specific language designed for specifying and verifying cryptographic algorithms. The Software Analysis Workbench (SAW) is a tool that allows developers to define the behavior of cryptographic algorithms and verify their implementations against these specifications. Cryptol uses symbolic execution to extract SAWCore models from C implementations, which are then verified against formal specifications written in Cryptol. SAW has been used to verify various cryptographic implementations.

Hanson, Winters, Mercer, and Decker 2022 prove the functional correctness of the SHA-3-256 implementation in OpenSSL using Cryptol and SAW. Cryptol also utilizes heavy automation with various solvers and a modest proof effort. The authors demonstrate how OpenSSL implementation-specific optimizations can be addressed by utilizing equivalence proofs for lower-level building blocks and subsequently using Cryptol overrides to replace the low-level implementations with high-level specifications. They also go into the challenge of converting a human-readable Specification into Cryptol code.

Compared to our work, Cryptol/SAW also focuses on proving an existing implementation. They, however, focus on functional correctness. In contrast, we focus on panic freedom and memory safety. Cryptol/SAW utilizes symbolic execution to extract models from C code. In comparison, HAX uses multiple phases of functional conversions to translate Rust code into F* code for verification.

6.4 SHA-2 and Verified Software Toolchain

The Verified Software Toolchain (VST) is a framework designed to verify the correctness of C programs through formal methods. It leverages Rocq, a general-purpose proof assistant, to provide machine-checked proofs of program correctness. Rocq allows developers to write formal proofs in a higher-order logic, ensuring that the proofs are both correct and complete. This framework is beneficial for verifying cryptographic implementations.

In their work, Appel 2016 present a formal verification of the SHA-256 implementation from the OpenSSL distribution using VST and Rocq. They provide a machine-checked proof of functional correctness, demonstrating that the implementation conforms to its specification. The verification process involves translating the C code into a form that can be reasoned about in Rocq, and then proving that the implementation meets the specified requirements. This approach ensures that the implementation is correct with respect to its functional specification.

Compared to our work, Appel focuses on the functional correctness of the SHA-256 implementation. Additionally, Appel uses the verified C compiler to achieve end-to-end verification. Both work with production implementations of hash functions and not toy examples.

7 Conclusion

This thesis aims to evaluate whether cryptography in Rust can be verified with modest effort and without compromising speed. We focused on SHA-3 and used HAX with F* as our toolchain. The core goal was panic freedom and termination for a production implementation, not toy examples. We also wanted to test a hypothesis based on experience: That round-based, buffer-oriented code can hide edge cases that testing misses, but formal methods can reveal.

We achieved the main objective. We have proven memory safety and panic freedom for SHA-3 paths on both the portable and SIMD backends. The SIMD work covers multi-lane absorption and squeezing and follows the same verification discipline as the scalar path. The verified code remains fast and idiomatic because only minimal changes were made (ρ and π permutations are split up and then are inlined again).

Formal methods paid off beyond the proof itself. During verification, we found an edge case in the XOF buffering API. Filling the internal buffer exactly to capacity and then calling absorb again left the buffer full, skipped absorption, and caused a later overflow and panic. The state invariant on the buffer length exposed the violation. A small change to check "buffer full" instead of "input consumed" fixes the bug. This is a realistic scenario that tests are unlikely to cover, and it confirms our intuition that buffer iteration can be a source of subtle errors.

We also distilled a small set of proof patterns that make this kind of code tractable. Large permutations are verified by taking a snapshot of the state and applying small helper functions, which avoids exponential growth in verification conditions. Loops rely on simple invariants, and we identified two lightweight arithmetic lemma to justify block-indexed access. Trait implementations mirror their pre- and postconditions to the trait interface. These techniques are easy to apply in Rust, map cleanly to F*, and are reusable by others.

There are limitations. Today, HAX has trouble modeling aggregates with mutable borrows, which complicates platform-specific variants for multi-lane code and adds duplication. Large, update-heavy functions remain sensitive to structure and require discipline in decomposition. Our strongest results are on panic freedom and termination. Full functional equivalence to a high-level Keccak specification was later deemed out of scope.

The path forward is clear. Prove functional equivalence for all multi-lane backends. Improve HAX ergonomics around diagnostics, traits, and mutable aggregates. Add F* tactics for routine steps such as loop bounds and slice-length preservation. Expand the

HAX library for bit-manipulation specifications and lemmas, enabling future projects to prove more with less effort.

The broader conclusion is simple: Verified cryptography in Rust is practical. With HAX and F*, we can build fast implementations that come with machine-checked guarantees, and we can transfer this method to other primitives beyond SHA-3.

List of Figures

List of Tables

Bibliography

Almeida, J. B., C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. Strub (2019). "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, pp. 1607–1622. ISBN: 9781450367479. DOI: 10.1145/3319535.3363211.

Appel, A. W. (Jan. 2016). *A Second Edition: Verification of a Cryptographic Primitive: SHA-256*. Tech. rep. A second edition; first edition appeared in emphTOPLAS 37(2):7:1–7:31 (April 2015). Accessed: today. Princeton University.

Beurdouche, B. (2020). "Formal verification for high assurance security software in FStar : application to communication protocols and cryptographic primitives". 2020UPSLE093. PhD thesis.

Bhargavan, K., M. Buyse, L. Franceschino, L. L. Hansen, F. Kiefer, J. Schneider-Bensch, and B. Spitters (2025). "hax: Verifying Security-Critical Rust Software Using Multiple Provers". In: *Verified Software. Theories, Tools and Experiments*. Ed. by J. Protzenko and A. Raad. Cham: Springer Nature Switzerland, pp. 96–119. ISBN: 978-3-031-86695-1.

Blanchet, B. (2025). *CryptoVerif*. https://bblanchet.gitlabpages.inria.fr/CryptoVerif/. Accessed: November 16, 2025. INRIA.

Blessing, J., M. A. Specter, and D. J. Weitzner (2021). "You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries". In: *CoRR* abs/2107.04940. arXiv: 2107.04940.

Butler, R. W., S. P. Miller, J. N. Potts, and V. A. Carreno (1998). "A Formal Methods Approach to the Analysis of Mode Confusion". In: *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*. IEEE.

Cremers, C., M. Horvat, S. Scott, and T. van der Merwe (2016). "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 470–485. DOI: 10.1109/SP.2016.35.

Cryspen (2022). *libcrux*. https://github.com/cryspen/libcrux. Accessed: November 16, 2025.

EasyCrypt (2025). https://www.easycrypt.info/. Accessed: November 16, 2025. EasyCrypt Project.

FIPS Publication 202: SHA-3 Standard: Permutation-based Hash and Extendable-Output Functions (Aug. 2015). Tech. rep. 202. Accessed: November 16, 2025. National Institute of Standards and Technology.

Hanson, P., B. Winters, E. Mercer, and B. Decker (2022). "Verifying the SHA-3 Implementation from OpenSSL with the Software Analysis Workbench". In: *Model Checking Software*. Ed. by O. Legunsen and G. Rosu. Cham: Springer International Publishing, pp. 97–113. ISBN: 978-3-031-15077-7.

Katz, J. and Y. Lindell (2007). *Introduction to Modern Cryptography: Principles and Protocols*. 1st ed. Chapman and Hall/CRC. DOI: 10.1201/9781420010756.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language*. 2nd ed. Prentice Hall.

Klabnik, S., C. Nichols, and T. R. Community (2019). *The Rust Programming Language*. https://doc.rust-lang.org/book/. No Starch Press.

Lean Theorem Prover (2025). https://lean-lang.org/. Accessed: November 16, 2025. Lean Community.

Martínez, G., D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy (2019). *Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms*. arXiv: 1803.06547 [cs.PL].

Matsakis, N. D. and F. S. Klock (2014). "The rust language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, pp. 103–104. ISBN: 9781450332170. DOI: 10.1145/2663171.2663188.

Mouha, N. and C. Celi (2023). *A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms*. Cryptology ePrint Archive, Paper 2023/331. DOI: 10.1007/978-3-031-30872-7_1.

National Security Agency, Cybersecurity and Infrastructure Security Agency, and Federal Bureau of Investigation (Nov. 2022). *The Software Memory Safety Cybersecurity Information Sheet*. Tech. rep. Accessed: November 16, 2025. National Security Agency.

Preneel, B. (2012). "The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition". In: *Conference Paper (COSIC Database)*. bart.preneel@esat.kuleuven.be. Katholieke Universiteit Leuven and IBBT, Dept. Electrical Engineering–ESAT/COSIC. Kasteelpark Arenberg 10 Bus 2446, B-3001 Leuven, Belgium.

Protzenko, J. and A. Raad (2024). "Verified Software". In.

RustCrypto Project (2017). *RustCrypto*. https://github.com/RustCrypto/. Accessed: November 16, 2025.

Stevens, M., E. Bursztein, P. Karpman, A. Albertini, and Y. Markov (2017). *The first collision for full SHA-1*. Cryptology ePrint Archive, Paper 2017/190.

The Heartbleed Bug (2014). https://heartbleed.com/. Accessed: November 16, 2025.

Verified Software Toolchain (2025). https://vst.cs.princeton.edu/. Accessed: November 16, 2025. Princeton University.

Wang, X. and H. Yu (2005). "How to Break MD5 and Other Hash Functions". In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by R. Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 19–35. ISBN: 978-3-540-32055-5.

Zinzindohoué, J.-K., K. Bhargavan, J. Protzenko, and B. Beurdouche (2017). "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, pp. 1789–1806. ISBN: 9781450349468. DOI: 10.1145/3133956.3134043.