Hard-to-Find Bugs in Cryptographic Software: Classification and Testing Methodologies

Abstract. Cryptographic software is essential for secure digital communication, yet its reliability can be compromised by subtle implementation flaws known as Hard-to-Find Bugs (HFBs). This survey paper examines the challenges posed by HFBs in cryptographic software, particularly within public-key cryptography, and explores a range of testing methodologies designed to detect them. We categorize HFBs, focusing on those arising from low-level implementation errors in modular and polynomial arithmetic. The paper analyzes the strengths and limitations of various testing techniques, including differential testing, static analysis, fuzzing, formal verification, and Known Answer Tests (KATs) specifically tailored to HFBs. For the latter we provide a C implementation for OpenSSL based on Wycheproof.

Keywords: Public Key Cryptography · Hard-to-Find Bugs · Software Testing · Formal Verification · Known Answer Tests

1 Introduction

Cryptographic software is foundational to secure digital communication, safeguarding the confidentiality, authenticity, and integrity of sensitive data. However, the security of these systems is intrinsically related to the correctness of their implementations. Subtle and elusive bugs, termed Hard-to-Find Bugs (HFBs), introduce critical vulnerabilities, which can lead to incorrect cryptographic computations, exploitable side-channel information leakage, or even key exposure. This paper[1] focuses on the challenges involved in identifying, analyzing, and mitigating HFBs in cryptographic implementations, with particular emphasis on public-key cryptography.

Classical asymmetric cryptography, with its reliance on complex mathematical operations such as long-integer modular arithmetic in RSA and ECC, is especially susceptible to implementation errors. These errors often arise due to the inherent complexity of algorithms, low-level implementation details, platform-specific constraints, and nuances within programming languages. Detecting HFBs is notoriously difficult because they typically emerge under rare conditions or in edge-case scenarios, evading conventional testing methods.

Although existing testing frameworks are vital for baseline correctness and regression testing, they are often insufficient to uncover the rare and subtle issues that typify HFBs. This paper bridges the gap by offering a comprehensive overview of testing methodologies designed specifically to target these elusive

[1] Full longer version available on anonymous GitHub repository: [https://anonymous.4open.science/r/wycheproof-c-3C6C/README.md].

bugs. We analyze the strengths and limitations of various techniques, including differential testing, static analysis, Monte Carlo testing, fuzzing, formal verification, and Known Answer Tests (KATs). Furthermore, we propose an integrated cryptographic testing framework that combines these methodologies to make the detection of HFBs better and easier.

The purpose of this research is to advance the security and reliability of cryptographic software by providing a detailed understanding of the challenges posed by HFBs and the tools available for their detection and mitigation and hereby closing a gap in the literature. By reviewing a wide array of testing methodologies and proposing a practical framework, this paper aims to contribute to the ongoing effort to ensure robust and trustworthy cryptographic implementations, thereby safeguarding the foundations of digital security.

2 Hard-To-Find Bugs (HFBs)

Hard-to-Find Bugs in cryptographic software are subtle errors that can compromise system security. These bugs are challenging to detect due to their low probability of occurrence and the specific conditions needed to trigger them—conditions that standard testing methods often fail to cover. Despite their rarity, HFBs can lead to serious vulnerabilities, such as incorrect cryptographic operations, side-channel vulnerabilities and private key leakage.

2.1 Characterization of HFBs

They are characterized by the following:

1. Low Probability of Occurrence: Only under rare and specific conditions, such as unique input combinations, specific sequences of operations, or particular hardware or timing factors.
2. High Complexity and Subtlety: Often result from complex system interactions, subtle hardware behaviors, or intricate algorithmic details, like improper handling of edge cases.
3. Difficulty in Detection and Reproduction: Conventional testing methods struggle to identify these bugs, and without knowing the trick they are hard to reproduce.
4. Cross-Disciplinary Nature: Addressing HFBs often requires expertise across multiple areas, including software, hardware, and cryptographic theory, due to their involvement in various layers of the technology stack.

Hard-to-find bugs (HFBs) are not confined to cryptographic software; they represent a widespread challenge across the technology sector. For example, Bressana et al. (2020) discuss the difficulty of detecting subtle data plane bugs in network hardware using their Portable Test Architecture (PTA), which revealed hidden issues such as performance degradation under specific traffic conditions [10]. Although their research focuses on network devices, the challenges they highlight closely mirror those encountered in cryptography: both fields are plagued by bugs that only manifest under rare conditions, evading traditional testing methods. In cryptographic software—particularly in public key cryptography (PKC), where low-level implementation errors in modular and polynomial arithmetic, along with other subtle bugs, can lead to catastrophic security breaches—the need for a comprehensive, rigorous verification framework becomes even more pressing. This paper aims to develop such a framework to systematically identify and mitigate these elusive vulnerabilities.

2.2 HFBs in PKC

Public key cryptography is especially prone to HFBs due to its reliance on complex mathematical operations, such as modular exponentiation in RSA and elliptic curve arithmetic in ECC. These operations demand precision, and even subtle implementation errors can create vulnerabilities that attackers can exploit. The public nature of PKC keys exacerbates this risk, allowing adversaries to craft malicious inputs and probe for flaws without needing privileged access.

A Detailed Example with the Sony PS3 Hack For example, in ECC the complexity of field arithmetic and point operations amplifies the potential for HFBs. Let's examine the Elliptic Curve Digital Signature Algorithm (ECDSA), where a signature is generated as follows:

1. Key Setup: Let G be a base point on an elliptic curve $E : y^2 = x^3 + ax + b \mod p$ with order n, and d be the private key. The public key is $Q = d \cdot G$.
2. Signature Generation:
   - Choose a random integer k such that $1 \leq k < n$.
   - Compute the point $(x_1, y_1) = k \cdot G$ using scalar multiplication.
   - Set $r = x_1 \mod n$ (if $r = 0$, restart with a new k).
   - Compute $s = k^{-1}(h + d \cdot r) \mod n$, where h is the hash of the message (if $s = 0$, restart).
   - The signature is the pair (r, s).
3. Verification: The verifier computes $u_1 = h \cdot s^{-1} \mod n$, $u_2 = r \cdot s^{-1} \mod n$, and checks if $(x_v, y_v) = u_1 \cdot G + u_2 \cdot Q$ satisfies $x_v \mod n = r$.

The security of ECDSA hinges on k being a fresh, unpredictable random number for each signature. A devastating HFB occurs if an implementation reuses the same k across multiple signatures—a flaw famously exploited in the Sony PlayStation 3 (PS3) hack in 2010 [16].

Suppose two signatures are generated:

- For message m_1 with hash h_1, signature (r, s_1) where $s_1 = k^{-1}(h_1 + d \cdot r) \mod n$.
- For message m_2 with hash h_2, signature (r, s_2) where $s_2 = k^{-1}(h_2 + d \cdot r) \mod n$.

If the same k is used, the r values are identical because $r = x_1 \mod n$ depends only on $k \cdot G$. An attacker can then:

– Subtract the equations:

$$s_1 - s_2 = k^{-1}(h_1 + d \cdot r) - k^{-1}(h_2 + d \cdot r) = k^{-1}(h_1 - h_2) \mod n$$

– Solve for k:

$$k = (h_1 - h_2)(s_1 - s_2)^{-1} \mod n$$

(assuming $s_1 - s_2 \neq 0 \mod n$, which is likely if $h_1 \neq h_2$).

– Recover the private key d from either signature, e.g.:

$$s_1 = k^{-1}(h_1 + d \cdot r) \implies s_1 \cdot k = h_1 + d \cdot r \implies d \cdot r = s_1 \cdot k - h_1$$

$$d = (s_1 \cdot k - h_1) \cdot r^{-1} \mod n$$

In the Sony PS3 case, Sony's ECDSA implementation used a fixed k value (or insufficiently random k), producing signatures with identical r values across different messages. Hackers collected these signatures, computed k, and recovered the private key, enabling them to sign unauthorized software and bypass security controls.

Why This is an HFB: This bug aligns with the characterization of HFBs outlined earlier:

1. Low Probability of Occurrence: The flaw manifests only when the same k is reused across signatures, a rare condition dependent on specific failures in the random number generator or a coding error that fixes k.
2. High Complexity and Subtlety: It arises from the intricate interplay of ECC arithmetic and randomness generation, a subtle detail in the implementation that is easily missed during standard development.
3. Difficulty in Detection and Reproduction: Conventional testing struggles to catch this without generating and comparing multiple signatures under varied conditions, and reproducing it requires knowing to check for repeated r values—a non-obvious trigger.
4. Cross-Disciplinary Nature: Addressing it demands expertise in cryptographic algorithms (ECC), software implementation (randomness handling), and hardware behavior (e.g., entropy sources), spanning multiple technical domains.

The mathematical precision of ECC operations means that such a subtle error in k can lead to a catastrophic security breach.

Broader Implications Both RSA and ECC demonstrate that PKC's reliance on complex arithmetic—modular exponentiation, field operations, and scalar multiplication—creates a fertile ground for HFBs. In RSA, errors in reduction or exponent handling can leak key bits; in ECC, flaws in randomness or point validation can expose the entire private key. Because attackers can interact with public components (e.g., sending crafted ciphertexts or observing signatures), they can amplify these bugs over time, making detection and mitigation in PKC implementations exceptionally challenging.

2.3 Impact and Exploitability

Impact The impact of bugs is assessed by the severity of the vulnerabilities they introduce, enabling developers to prioritize fixes based on potential damage. Key factors for evaluating impact include:

- Exposure of sensitive data
- System integrity compromise
- Exploitability by attackers
- Long-term implications for system security

This classification is crucial for understanding cryptographic risks and underscores the need for expert knowledge in developing secure systems.

HFBs pose a significant security risk due to their potential exploitability by attackers. Their elusive nature makes them prime targets for adversaries who can craft precise inputs or operation sequences to exploit them. This can lead to severe consequences, such as cryptographic failures, unauthorized data access, or even private key leakage, compromising the integrity and confidentiality of secure systems.

Exploitability Given their low probability of occurrence under normal conditions, such bugs can remain undetected for extended periods. A notable example is the ROCA vulnerability (CVE-2017-15361), which affected RSA key generation in Infineon chips used by OpenSSL and other cryptographic libraries. The flaw in the key generation algorithm allowed attackers to factorize the public key and recover the private key. Introduced around 2012, it remained undetected until its discovery in October 2017 by researchers from Masaryk University, Enigma Bridge, and Ca' Foscari University [25].

Moreover, there is an increasing risk of maliciously introduced HFBs, particularly in environments with frequent and large-scale code contributions, such as open-source projects. This risk is especially significant for malicious nation-state actors, who may seek to embed hidden backdoors in software widely regarded as secure, such as cryptographic libraries, to enable long-term espionage or control over critical systems. Attackers could intentionally introduce subtle HFBs during major commits or feature additions, which may evade detection during standard reviews. A recent example is the 2024 XZ Utils incident, where a malicious actor used social engineering to gain trust and insert a subtle backdoor into the software, potentially allowing unauthorized code execution on debian machines [29]. These flaws are often masked within complex or large code changes, especially in low-level languages like C, C++, and Assembly, which are commonly used in cryptography implementations for performance reasons. Assembly, in particular, is still prevalent for critical optimizations, adding to the challenge of detecting such bugs.

2.4 Hardware-Induced Errors

While our focus is on software HFBs, cryptographic failures can also stem from hardware-induced errors, which are often indistinguishable from software faults.

Transient faults—such as bit flips in RAM or registers caused by cosmic radiation or electrical interference—can introduce security risks exploitable via fault injection attacks, as explored in comprehensive analyses of natural and induced faults [22]. Techniques like voltage glitching and electromagnetic fault injection can deliberately induce errors, potentially leaking sensitive data, such as private keys through faulty signature generation, as demonstrated in theoretical models of random hardware faults [9].

Though mitigating hardware-induced errors is a crucial research area, it is beyond the scope of this study. Addressing these risks requires error detection mechanisms, hardware-level protections, and resilient cryptographic implementations.

3 Cryptography Testing Techniques

Hard-to-Find Bugs in cryptographic implementations pose a significant challenge due to their infrequent occurrence and subtle nature. This section provides a structured analysis of existing testing methodologies designed to uncover these elusive defects. While some verified implementations can guarantee the absence of certain HFBs, no single approach is universally effective across all cryptographic systems. We evaluate various methodologies—including differential testing, formal verification, fuzzing, and Known Answer Tests (KATs)—for their effectiveness in identifying critical vulnerabilities. Based on this assessment, we propose a standardized framework for robust cryptographic testing.

3.1 Differential Testing

Differential testing applies identical inputs to multiple independent implementations of a algorithm or function to detect inconsistencies. Discrepancies in outputs indicate potential bugs caused by errors in arithmetic, parameter handling, or edge-case logic.

This approach is particularly valuable for identifying hard-to-find bugs in complex systems, such as cryptographic software, where exhaustive specification-based testing may be impractical due to the vastness of key spaces, high entropy, and probabilistic behaviors.

For cryptographic software, this might include feeding identical plaintexts and keys to different RSA implementations (e.g., OpenSSL vs. BoringSSL vs. Cryptlib) and checking for consistency in ciphertexts or key generation outcomes.

Limitations Differential testing is effective for identifying subtle errors by comparing multiple implementations without requiring a formal specification, assuming independent implementations are unlikely to share the same flaws. While its good to indentify some flaws, it is less suited for detecting those that occur in very rare edge cases such as HFBs. Instead, it is more useful for confirming bugs once discrepancies arise. Its effectiveness also depends on the availability of multiple implementations and the assumption that differences indicate bugs rather than intentional design variations.

3.2 Static Analysis

Static analysis refers to a broad set of techniques that analyze source code, bytecode, or binaries without execution to detect potential errors, vulnerabilities, and inefficiencies. These methods range from simpler approaches, such as linting and type checking, to more advanced techniques like abstract interpretation and methods to translate source code into mathematical models.

In cryptographic software, static analysis plays a key role in verifying security properties. For example, formal verification tools like Cryptol translate cryptographic algorithms into Satisfiability Modulo Theories (SMT) formulas (often with boolean satisfyability), enabling automated correctness proofs. Other tools, such as ct-verif, focus specifically on verifying constant-time execution, while binary-level analysis platforms like BinSec detect vulnerabilities in compiled cryptographic binaries.

Example: RSA signature In the case of RSA signatures, static analysis ensures that the implementation correctly performs modular exponentiation and follows the expected structure.

Given a message m, the RSA-PSS signature s is computed as:

$$s = (H(m) + \text{salt})^d \mod N \tag{1}$$

A static analysis tool encodes both the high-level specification and the low-level implementation as SMT formulas. The verification process ensures that the computed s follows the expected structure across all valid inputs. If a discrepancy arises, it indicates a potential bug in arithmetic operations, bitwise manipulations, or edge-case handling.

The following Galois' Cryptol specification represents the RSA-PSS signature generation:

```
rsaPssSign : {n} (fin n) => [n] -> [h] -> [saltLen] -> [n]
rsaPssSign d hash salt = modExp (hash + salt) d n
```

where `d` is the private key, `hash` represents the hashed message, `salt` is a random value, and `modExp` performs modular exponentiation.

To verify correctness, a static analysis tool translates both this Cryptol specification and its corresponding low-level implementation into SMT formulas. An SMT solver, such as Z3 or CVC5, then attempts to prove that the implementation adheres to the mathematical definition. The solver does this by:

– Encoding modular exponentiation and arithmetic constraints as bitvector logic.
– Checking that the computed signature is always valid under all possible inputs.
– Finding counterexamples if an invalid transformation exists.

If the SMT solver detects a discrepancy, it provides a counterexample, helping identify potential bugs related to arithmetic overflow, incorrect padding, or timing side channels.

Integration Galois' Cryptol serves as a prominent specification and verification platform that translates cryptographic functions into SMT formulas for analysis by solvers. A notable application of this approach is Amazon Web Services' (AWS) formal verification of their s2n TLS/SSL implementation. By employing Cryptol and associated tools, AWS identified and mitigated potential vulnerabilities, such as timing side-channel attacks in HMAC, prior to deployment [12].

Limitations Static analysis excels at detecting logical errors, boundary issues, and non-constant-time behaviors that dynamic testing might overlook. By exhaustively verifying properties across all inputs, it offers strong assurance, making it particularly suitable for uncovering subtle logical flaws or timing variations that could compromise security. However, the effectiveness of static analysis is contingent upon the accuracy and completeness of the specifications. As noted, if the specification is correct, a proven code should adhere to it. Nonetheless, subtle bugs might lurk within the definition itself, underscoring the importance of meticulous specification design. This challenge is particularly evident when ensuring the randomness properties of cryptographic components, where incomplete or flawed specifications could fail to capture statistical weaknesses detectable through rigorous testing suites, such as those outlined in NIST statistical test suite for random and pseudorandom number generators, which could be integrated with SMT-based analysis to enhance verification robustness [27].

3.3 Monte Carlo Testing

Monte Carlo Testing uses randomness to assess the reliability of cryptographic algorithms. Unlike deterministic methods with fixed test cases, it samples inputs randomly from a probability distribution P over the input domain $\mathcal{X}$, simulating diverse real-world conditions to uncover subtle bugs like arithmetic overflows or implementation flaws.

For a cryptographic function $f : \mathcal{X} \to \mathcal{Y}$, the test draws n random inputs $x_i \sim P$, computes outputs $y_i = f(x_i)$, and statistically analyzes the set $\{y_1, \ldots, y_n\}$ for properties like uniformity or unexpected patterns. The choice of P—e.g., uniform for broad coverage or biased toward edge cases like malformed inputs—depends on the algorithm and suspected vulnerabilities, typically implemented via a high-quality pseudorandom number generator (PRNG).

For example, testing a hash function $H : \{0,1\}^* \to \{0,1\}^{256}$ might sample $n = 10^6$ messages from a uniform P over lengths 0 to 1024 bytes. Hash values $H(x_i)$ are checked for collisions (probability $\approx \frac{n^2}{2^{257}}$) or skew via a chi-squared test. Deviations could signal flaws, such as PRNG bias, with P adjustable to target specific weaknesses like edge-case handling.

Integration Monte Carlo Testing excels at uncovering rare inconsistencies that arise from errors in modular arithmetic, improper edge-case handling, or inadequate parameter validation. Its strength lies in its flexibility: by tuning P and increasing n, the test can probe deeper into the input space, offering broader statistical coverage than deterministic methods alone.

Limitations However, its effectiveness depends heavily on the choice of P. A poorly chosen distribution might miss critical regions of $\mathcal{X}$, failing to trigger certain flaws. Moreover, its probabilistic nature means it cannot guarantee the detection of all issues, especially those tied to highly specific inputs or side-channel attacks (e.g., timing leaks).

3.4 Fuzzing

Fuzzing is an automated testing technique that discovers implementation flaws by generating malformed or adversarial inputs to stress-test cryptographic systems. Fuzzing empirically uncovers vulnerabilities like buffer overflows, input validation errors, and protocol deviations.

Application to Cryptographic Testing Given a cryptographic function $f : \mathcal{X} \to \mathcal{Y}$, fuzzing generates inputs $x_i \in \mathcal{X}^*$, where $\mathcal{X}^*$ represents an extended set that includes malformed or out-of-specification inputs. The function outputs $y_i = f(x_i)$ are evaluated to detect anomalies, such as:

$$\text{Anomaly} \quad \text{if} \quad y_i \notin \mathcal{Y} \quad \text{or} \quad \text{behavior}(f(x_i)) \neq \text{expected}.$$

Anomalies may include crashes, incorrect results, or abnormal execution behavior, indicating potential bugs. There are different fuzzing approaches:

- Mutation-Based: Modifies existing inputs (e.g., flipping bits in X.509 certificates) using heuristics.
- Generation-Based: Constructs inputs from formal grammars (e.g., TLS message syntax [21]).
- Coverage-Guided: Prioritizes inputs that explore new code paths (e.g., AFL [34], libFuzzer).
- Differential: Compares outputs across implementations given identical inputs to detect inconsistencies [11].

Limitations However, fuzzing has inherent limitations:

- Logical Flaws: Cannot detect incorrect arithmetic (e.g., missing modular reductions) without oracles.
- Side Channels: Timing variations or memory access patterns require specialized tools like ctgrind.
- Statefulness: Multi-step protocols (e.g., TLS handshakes) need state-aware fuzzers [14].

A prominent example of large-scale fuzzing is OSS-Fuzz [35], an open-source framework developed by Google. By continuously fuzzing cryptographic libraries like OpenSSL, BoringSSL, and Libsodium, OSS-Fuzz has identified numerous critical bugs, enhancing the security of widely used cryptographic implementations.

Since 2016, OSS-Fuzz has identified 500+ cryptographic bugs, including, CVE-2020-1968 and CVE-2021-3449: NULL pointer dereference in OpenSSL ASN.1 parsing.

These findings underscore fuzzing's role in hardening real-world implementations, though it must be augmented with other testing methods for full assurance for full assurance.

3.5 Known Answer Tests (KATs)

KATs are a fundamental method for validating cryptographic implementations by comparing their output against reference values (test vectors) derived from trusted implementations. They ensure that low-level optimizations or hardware-specific modifications do not alter the core behavior of the algorithm. They are widely employed in the validation of classical and post-quantum algorithms, including in the NIST standardization process of algorithms where some are always provided.

Given a cryptographic function $f : \mathcal{X} \to \mathcal{Y}$ and a finite set of known input-output pairs $\{(x_i, y_i) \mid i = 1, \ldots, n\}$, where $y_i = f(x_i)$ has been computed using a reference model, a KAT verifies that an optimized implementation f' satisfies:

$$f'(x_i) = y_i \quad \forall i = 1, \ldots, n.$$

This ensures that f' preserves the correctness of f across the selected inputs, thereby confirming that code-level changes, such as pipeline rearrangements or hardware optimizations, have not introduced functional errors.

Example: ECDSA Validation Consider ECDSA, where the goal is to validate signature generation and verification. ECDSA involves generating a signature pair (r, s) for a given message m using a private key d and a generator point G on the elliptic curve E. The signature is computed as:

$$r = (kG)_x \mod n, \quad s = k^{-1}(H(m) + dr) \mod n,$$

where k is a random nonce, n is the order of G, and H is a cryptographic hash function.

For a given set of input messages $\{m_1, m_2, \ldots, m_k\}$ and corresponding reference signatures $\{(r_i, s_i) \mid i = 1, \ldots, k\}$, a KAT ensures that the implementation produces the correct signature pair:

$$(r'_i, s'_i) = \text{Sign}(d, m_i) \quad \forall i = 1, \ldots, k, \quad \text{where} \quad (r'_i, s'_i) = (r_i, s_i).$$

Any deviation in the output indicates potential issues in the implementation, such as incorrect handling of modular arithmetic, nonce generation, or elliptic curve point operations. Such discrepancies can lead to incorrect or insecure signatures.

Verification involves checking that the generated signature pair satisfies the ECDSA verification equation:

$$(r'_i, s'_i) \quad \text{valid} \iff r'_i = (u_1 G + u_2 Q)_x \mod n,$$

where $u_1 = s^{-1}H(m) \mod n$ and $u_2 = s^{-1}r \mod n$, ensuring that signature correctness is maintained under all valid inputs.

Wycheproof Wycheproof [31], an open-source testing library by Google, targets subtle vulnerabilities in cryptographic software. It offers a comprehensive suite of Known Answer Tests (KATs), timing tests, and unit tests to uncover Hard-to-Find Bugs (HFBs) and edge cases in cryptographic primitives. While effective against known weaknesses, it does not prove a cryptosystem secure, as it relies on predefined cases.

Originally Java-based, Wycheproof suits libraries like Bouncy Castle. We provide an extended C implementation of the public key testing in Wycheproof along with this paper for testing C-based libraries (e.g., OpenSSL), available on GitHub [2]. This port enhances accessibility without altering its core methodology.

Limitations of KATs While passing KATs offers reassurance that the essential mapping from inputs to outputs remains unchanged, it does not guarantee security under all conditions. KATs do not explore the full input space, nor do they inherently detect timing anomalies or subtle side-channel issues. They should be complemented with more probabilistic or property-based testing methods, as well as formal verification and fuzzing, to achieve broad and reliable assurance.

In practice, cryptographic testing should integrate KATs into a larger verification framework that includes additional forms of analysis.

3.6 Framework for Cryptographic Testing

Given the complexity and sensitivity of cryptographic implementations, a robust testing framework should employ a combination of deterministic, probabilistic, and dynamic methods to ensure both correctness and resilience against vulnerabilities. A practical approach for identifying Hard-to-Find Bugs with relatively low computational resources involves leveraging three key techniques: Known Answer Tests (KATs), Differential Testing, and Continuous Fuzzing.

Proposed Techniques

Differential Testing Differential Testing compares outputs from multiple independent implementations given identical inputs. Discrepancies signal potential errors caused by differences in arithmetic operations, parameter handling, or edge-case logic. While it excels at identifying mismatches, it does not inherently target rare HFBs unless combined with complementary methods like fuzzing.

Known Answer Tests (KATs) KATs validate correctness by comparing outputs against known reference values. They provide a reliable baseline for detecting deviations introduced by optimizations or hardware-specific changes. Libraries like Wycheproof extend KATs by targeting known edge cases and previously reported HFBs. Despite their utility, KATs cannot detect new or unknown bugs without additional testing strategies.

Continuous Fuzzing Continuous fuzzing generates malformed or edge-case inputs to test robustness. Unlike Monte Carlo Testing, which operates on valid input distributions, fuzzing applies random mutations to uncover unexpected failures. Tools like OSS-Fuzz have proven effective by continuously fuzzing cryptographic libraries (e.g., OpenSSL, Libsodium), identifying critical bugs that deterministic methods might overlook.

Integration of Techniques and Limitations A robust cryptographic testing framework integrates Differential Testing, KATs, and Fuzzing:

- Differential Testing highlights discrepancies across multiple implementations.
- KATs ensure correctness for well-defined inputs.
- Continuous Fuzzing stresses robustness under malformed and edge-case inputs.

While this combination addresses many common and known vulnerabilities, it has inherent limitations:

- Coverage: KATs and Differential Testing are limited by finite input sets and cannot explore every possible scenario, especially new ones.
- Heuristics dependency: The success of fuzzing depends on the mutation strategy and campaign duration. It may fail to detect logical flaws or vulnerabilities requiring formal analysis.

To achieve comprehensive assurance, these techniques should be complemented by formal verification, which offers mathematical guarantees for correctness and security properties. Together, they form an effective, layered approach to cryptographic testing, balancing empirical validation with theoretical rigor.

4 Verified Cryptographic Implementations

Ensuring the correctness and security of cryptographic implementations is paramount. This section delves into formal verification methods and the utilization of programming languages like Rust and Jasmin to achieve high-assurance cryptographic software.

4.1 Formal Verification of Cryptographic Software

Formal verification applies mathematical logic to rigorously prove that cryptographic software adheres to formally specified correctness and security properties across all possible executions. Unlike conventional testing, which inspects only a finite subset of inputs, formal verification provides global guarantees by exhaustively analyzing every possible execution path. This process ensures the absence of implementation flaws such as arithmetic errors, incorrect memory handling, and logic inconsistencies that could compromise security [4, 8].

The verification process begins with the formal specification of a cryptographic function $f : \mathcal{X} \to \mathcal{Y}$ and its desired properties. For a public-key encryption scheme, correctness is typically expressed as:

$$\forall (pk, sk), \forall m \in \mathcal{M}, \quad \mathrm{Dec}(sk, \mathrm{Enc}(pk, m)) = m.$$

This property, a global invariant, must hold under all conditions. Verification tools such as Coq, EasyCrypt, and Tamarin Prover translate both the formal specification and implementation into logical formulas, using first-order or higher-order logic, and attempt to establish their equivalence through automated theorem proving or model checking [5].

Formal Verification of Cryptographic Protocols Formal verification is extensively applied to cryptographic protocols, ensuring properties such as confidentiality, integrity, and authentication. One common approach uses the Dolev-Yao model, which abstracts cryptographic primitives as perfect black boxes while modeling adversarial interactions. Tools like ProVerif and Tamarin Prover automate the verification process, analyzing protocol logic to detect vulnerabilities. For instance, ProVerif has been instrumental in verifying the security of the Transport Layer Security (TLS) protocol, identifying weaknesses and validating security patches [8].

Formal verification has also been applied to post-quantum cryptographic schemes, ensuring resistance to quantum adversaries. Works such as [3] have employed EasyCrypt to analyze lattice-based cryptographic primitives, verifying their security under standard assumptions.

Challenges and Limitations Despite its strengths, formal verification faces several challenges. The complexity of cryptographic algorithms and protocols often leads to state-space explosion, making exhaustive analysis computationally expensive. Additionally, ensuring that the formal model accurately represents real-world implementations is critical, as discrepancies can lead to undetected vulnerabilities. Furthermore, formal methods struggle to capture side-channel attacks and implementation-specific optimizations, necessitating complementary empirical validation techniques such as fuzz testing and differential analysis [6].

While formal verification is a powerful tool for proving the correctness of cryptographic implementations, it is most effective when integrated with empirical testing to achieve comprehensive security validation.

4.2 Memory Safety and Concurrency

Memory safety is paramount in software, as vulnerabilities can lead to severe security breaches. A significant portion of software vulnerabilities, approximately 70%, stem from memory safety issues in languages like C, C++, and Assembly [26]. The NSA report advocates for the adoption of memory-safe languages to mitigate these risks, particularly in security-critical.

Rust emerges as a strong candidate due to its robust memory safety guarantees and performance capabilities. Rust's ownership and borrowing system prevents common memory errors such as null pointer dereferences, buffer overflows, and data races. This is crucial for cryptographic functions, where any unintended memory access can compromise security. Formally, Rust's borrow checker ensures that for any memory state $\mathcal{M}$, a function f will execute safely:

$$\forall \mathcal{M}, \quad \text{BorrowCheck}(f, \mathcal{M}) = \text{safe}$$

This guarantee is crucial for cryptographic code, where unintended memory access can lead to security breaches. Moreover, Rust's type system enforces that operations are performed on compatible types, reducing the risk of logical errors in cryptographic algorithms.

Furthermore, Rust's type system enforces type compatibility, reducing the likelihood of logical errors within cryptographic algorithms. Libraries like 'subtle' facilitate the development of constant-time code, essential for preventing timing attacks by ensuring operations exhibit no data-dependent timing variations.

Several cryptographic projects have already leveraged Rust's safety features. For instance the 'ring' [30], 'rustls' [7], and 'RustCrypto' [33] crates. Alternatives like Go or Java sacrifice performance for garbage collection, making them unsuitable for low-latency applications [17].

Rust typically exhibits a slight performance penalty compared to C in cryptographic operations, yet it remains highly competitive with aggressive optimization. Benchmarks, such as those for AES implementations, demonstrate Rust nearing C's performance, with any overhead largely stemming from higher-level abstractions rather than intrinsic limitations [28]. In contrast, C's minimalism often provides an advantage in resource-constrained settings like embedded systems [23]. While C excels at low-level, byte-by-byte efficiency, Rust's richer abstractions, simplified memory management, and extensive libraries can inflate code size and complexity if unoptimized, though its optimization potential occasionally surpasses C's by leveraging advanced function and library integration. Furthermore, Rust supports formal verification through tools like Libcrux, a cryptographic library integrating verified artifacts via hacspec for correctness and security proofs, and Hax, a toolchain translating Rust into formal languages (e.g., Coq, F*) for security-critical applications [18, 24]—capabilities absent in C, which lacks direct verification support. Thus, selecting between Rust's safety and C's raw performance requires careful trade-off analysis for cryptographic implementations.

4.3 Low-Level Optimization and Verification

Jasmin [1] bridges high-level cryptographic specifications and verified assembly implementations through a certified compiler (`jasminc`) and formal verification in EasyCrypt. Its syntax mirrors low-level control flow while enabling mathematical reasoning about correctness and side-channel resistance. The methodology to formally verify a code with jasmin is simple:

Methodology

1. Modeling: Translate assembly to Jasmin, preserving low-level optimizations.
2. Annotation: Insert pre/post-conditions and loop invariants using EasyCrypt predicates.
3. Equivalence Checking: Use the CryptoLine toolchain to prove equivalence between Jasmin code and reference models via algebraic predicates (e.g., $\texttt{mont\_asm}(x) \equiv \texttt{mont\_ref}(x) \mod p$).
4. Compiler Certification: Rely on `jasminc`'s correctness to ensure compiled assembly matches Jasmin semantics.

This approach combines automated SMT solving (for range checks) and interactive theorem proving (for modular equivalences), enabling verification of both arithmetic correctness and side-channel security in lattice-based cryptography.

We demonstrate its application to two critical tasks: verifying constant-time arithmetic and equivalence proofs between optimized code and reference models.

Verified Montgomery Reduction. Consider the Montgomery reduction algorithm for prime modulus p, where $R = 2^{64} > p$ and $p' = -p^{-1} \mod R$. A Jasmin implementation computes $\text{Mont}(x) = (x + (x \cdot p' \mod R) \cdot p)/R$, which must satisfy:

$$\text{Mont}(x) \equiv xR^{-1} \mod p \quad \text{and} \quad 0 \leq \text{Mont}(x) < 2p$$

The Jasmin code below computes this in assembly-optimized form, with annotations for verification:

```
fn montgomery (uint64 x0, x1) -> (uint64 r) {
    %rax = x0
    mulx %rax, %rdx, %rax      # x * p' (low)
    mulx p, %rdx, %rax         # (x * p') * p
    addc %rax, x0, %rax        # x + (x * p' mod R) * p
    adc %rdx, x1, %rdx
    shr $64, %rax, %rdx        # >> 64 (division by R)
    assert @rpre (x0 + x1*2^64 < 2^128);
    assert @rpost (result == (x0 + x1*2^64)*R^{-1} mod p);
    return %rdx
}
```

Here, `assert` clauses specify pre/post-conditions. EasyCrypt proves the equivalence between this code and the mathematical specification using symbolic execution and modular arithmetic lemmas. The compiler guarantees constant-timeness by rejecting branching on secrets.

Equivalence Proofs in Falcon's Floating-Point Arithmetic. In [19], Jasmin verified a critical discrepancy in Falcon's emulated floating-point multiplication. The original code zeroized products prematurely based on exponent thresholds, violating IEEE 754 rounding rules. Let $x = (-1)^{s_x} 2^{e_x - 1075}(2^{52}+m_x)$, $y = (-1)^{s_y} 2^{e_y - 1075}(2^{52} + m_y)$. The faulty implementation computed:

$$z_{\text{emu}} = \begin{cases} 0 & \text{if } e_x + e_y - 2100 < -1076 \text{Round}(x \cdot y) \\ \text{otherwise} & \end{cases}$$

where the zeroization (Line 16 in Algorithm 4) preceded rounding. Jasmin was used to implement a corrected version where zeroization occurs after rounding. The equivalence proof required showing:

$$\forall x, y \in \text{FP}_{normal}, \quad z_{\text{emu}} \neq z_{\text{correct}} \implies \exists k \in Z, |x \cdot y - k \cdot 2^{e_z}| \leq 2^{e_z - 53}$$

By compiling both versions to assembly and verifying their equivalence against a CryptoLine model, the authors demonstrated that 692/2048 FFT constants in Falcon could trigger incorrect zeroization, yet the error was provably absent in the FFT due to lower bounds on intermediate values.

4.4 Summary

The combination of formal verification methods with programming languages designed for safety and performance, such as Rust and Jasmin, provides a robust foundation for developing secure cryptographic implementations. Rust's guarantees of memory safety and concurrency, and Jasmin's support for low-level optimization and formal verification, provide developers with the ability to build cryptographic software that is both efficient and provably secure. Using these methods when needed along with cryptographic testing methods presented in the previous section 3.6 constitute the state of the art for insurance of cryptogaphic software.

5 Bugs Collection

This section systematically classifies hard-to-find bugs in cryptographic implementations, drawing from vulnerabilities identified in open-source projects, standardization submissions, advanced testing methodologies, and databases like CVE. These bugs often elude conventional testing and can undermine security by mishandling core mathematical operations; however, not all result in severe vulnerabilities, as their impact is frequently mitigated by system redundancies and the absence of single-point-of-failure dependencies.

Each bug class is rigorously defined with:

- `CARRY_PROPAGATION.md`: Carry chain mismanagement in multi-precision arithmetic
- `CRYPTO_STATE.md`: Incorrect updates to secret-dependent program states

- `IMPLEMENTATIONS.md`: Algorithmic deviations from mathematical specifications
- `INPUT_VALIDATION.md`: Boundary condition oversights in adversarial inputs
- `PARAM_HANDLING.md`: Coefficient encoding/decoding flaws in structured lattices
- `CONSTANT_TIME.md`: Secret-dependent branches/memory accesses

Each of the bug is then documented using the following format:

1. `Specification`: Description of the bug's context and affected component(s).
2. `Defect`: The exact nature of the defect and the error introduced by it.
3. `Impact`: The consequences of the defect, particularly in terms of cryptographic correctness and potential security risks.
4. `Code Snippet`: A sample of the code or assembly where the bug manifests.

This structured documentation helps us better understand the bugs, trace their origins, and identify patterns that can improve future testing and verification efforts.

5.1 Carry Propagation Errors

Carry propagation errors are a specific class of arithmetic bugs that occur during multi-precision operations, where intermediate carries generated by addition, subtraction, or multiplication operations are not correctly propagated across word boundaries.

Consider two multi-precision integers A and B, each represented as an array of n machine words $A = [a_0, a_1, \ldots, a_{n-1}]$ and $B = [b_0, b_1, \ldots, b_{n-1}]$, where each word has a fixed bit width w (e.g., 64 bits). Adding these integers involves computing:

$$R_i = a_i + b_i + \text{carry}_{i-1},$$

where carry_{i-1} is the carry from the previous word addition. If the sum exceeds $2^w - 1$, a carry is generated:

$$\text{carry}_i = \begin{cases} 1 & \text{if } R_i \geq 2^w \\ 0 & \text{otherwise.} \end{cases}$$

Failure to correctly propagate this carry across all words results in incorrect final values.

A way to completely avoid this kind of problem is to use smaller words:

5 Limbs Over 4 In Curve25519 implementations (e.g., Ed25519 per RFC 8032), field elements modulo $2^{255} - 19$ can be represented using four 64-bit limbs (256 bits) or five 51-bit limbs (255 bits). The unsaturated approach with 51-bit limbs, adopted by libraries like libsodium [15] and curve25519-dalek's [20] serial backend, limits each limb to $< 2^{51}$. For addition, $a_i + b_i < 2^{52} < 2^{64}$, avoiding carry propagation within a 64-bit word. Multiplication, however, yields a 102-bit product ($a_i \cdot b_i < 2^{102}$), split into two 64-bit words: a low part $< 2^{64}$ and a high part $< 2^{38}$ ($\lfloor(2^{51}-1)^2/2^{64}\rfloor$).

Using five limbs increases multiplications from 16 (4×4) to 25 (5×5), but the smaller high parts of partial products ($< 2^{38}$) reduce carry propagation. On RISC-V (e.g., SiFive U74), where add-with-carry takes 2.5 cycles, crrl's m51 backend pre-shifts limbs (e.g., $a_0 \ll 6$), aligning products to word boundaries and eliminating carry operations (158928 vs. 202021 sign cycles for m64). Conversely, on x86 (e.g., Coffee Lake), efficient ADCX/ADOX instructions make four 64-bit limbs faster (49130 vs. 70149 sign cycles for m51) [32]. While the unsaturated approach trades more operations for reduced carry overhead—in most cases less efficient—its complete removal of carries offers a key advantage in our case: it eliminates carry propagation errors, which account for roughly 30% of hard-to-detect bugs in our classification, prioritizing reliability over raw performance.

Assembly-Level Carry Propagation Bug [CVE-2017-3736]

Background This vulnerability pertains pertains to a carry propagation flaw in the Montgomery squaring routine within the OpenSSL cryptographic library. This vulnerability affects systems running on x86_64 architecture, particularly those with processors supporting BMI1, BMI2, and ADX extensions, such as Intel Broadwell and AMD Ryzen. The Montgomery squaring routine, implemented in assembly, is a critical part of the elliptic curve cryptography operations over the NIST P-256 curve (Weierstrass named secp256r1 in OpenSSL): Defined by $y^2 = x^3 - 3x + b \mod p$, where $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Mathematical Context Montgomery multiplication is used to efficiently compute modular products without explicitly performing costly division operations. Given two large integers a and b, and a modulus N, Montgomery multiplication computes:

$$\text{Mont}(a, b, N, R) = (a \cdot b \cdot R^{-1}) \mod N,$$

where R is a power of 2 larger than N. The result of Montgomery multiplication is in a transformed domain, and repeated applications (e.g., squaring) require proper handling of intermediate carries to ensure correctness.

In the specific case of Montgomery squaring, given an input a, the goal is to compute:

$$a^2 \cdot R^{-1} \mod N.$$

This operation involves multi-precision multiplication of a with itself, followed by reduction modulo N using Montgomery's method.

Defect Description The defect in CVE-2017-3736 arises during the multi-precision addition step in the Montgomery squaring routine. Specifically, the assembly code fails to correctly propagate carries across word boundaries during the accumulation of partial products. This results in incorrect intermediate values, leading to an incorrect final result.

Consider a simplified example of multi-precision addition where two 64-bit words are added:

$$(r_0, r_1) = (a_0 + b_0, a_1 + b_1 + \text{carry}),$$

where r_0 and r_1 are the resulting words, and carry is generated if the sum of a_0 and b_0 exceeds the word size (64 bits).

In the flawed implementation, the assembly code neglects to properly handle the carry propagation during this accumulation step. The relevant portion of the buggy code is:

```
.align  32
.Lsqrx8x_break:
  sub    16+8(%rsp), %r8         # consume last carry
  xor    %ebp, %ebp              # clear %ebp
  adc    $0, %r9                 # add carry to next word
  adc    $0, %r10
  adc    $0, %r11
  adc    $0, %r12
  adc    $0, %r13
  adc    $0, %r14
  adc    $0, %r15
  cmp    $carry, $tptr           # compare carry
  je     .Lsqrx8x_outer_loop
```

Analysis of the Error In the above code, the subtraction instruction `sub 16+8(%rsp), %r8` is intended to consume the last carry from a previous addition. However, due to a missing `adc` instruction in the subtraction sequence, the carry propagation chain is disrupted, leading to incorrect results in the subsequent additions. The `adc` (add with carry) instruction is essential for correctly propagating carries across word boundaries.

To illustrate this, assume the following intermediate values:

– Initial carry from a previous operation: $C = 1$.
– Partial products to be accumulated: $P_0, P_1, \ldots, P_7$.

The correct accumulation should proceed as:

$$R_0 = P_0 + C,$$

$$R_1 = P_1 + \text{carry from } R_0,$$

$$R_2 = P_2 + \text{carry from } R_1,$$

and so on. The missing `adc` instruction prevents the carry from being correctly propagated beyond R_0, resulting in incorrect values for $R_1, R_2, \ldots$.

Impact This defect impacts the correctness of elliptic curve scalar multiplication, which relies on repeated Montgomery squaring. An incorrect result in this computation can lead to invalid elliptic curve points being generated, potentially compromising the security of cryptographic protocols such as ECDSA and ECDH which directly use this weierstrass curve and it's montgomery squaring.

Fixed Code The issue was resolved by ensuring that the carry propagation chain is correctly maintained. The corrected code includes an additional `adc` instruction to handle the carry properly:

```
.align  32
.Lsqrx8x_break:
  sub    16+8(%rsp), %r8          # consume last carry
  adc    $0, %r8                  # propagate carry correctly
  xor    %ebp, %ebp               # clear %ebp
  adc    $0, %r9                  # add carry to next word
  adc    $0, %r10
  adc    $0, %r11
  adc    $0, %r12
  adc    $0, %r13
  adc    $0, %r14
  adc    $0, %r15
  cmp    $carry, $tptr            # compare carry
  je     .Lsqrx8x_outer_loop
```

By adding the missing `adc` instruction, the carry chain is correctly propagated, ensuring the accuracy of the Montgomery squaring operation.

Carry propagation errors, such as CVE-2017-3736, highlight the complexity and fragility of low-level cryptographic implementations. Even minor oversights in handling carries can result in severe vulnerabilities. This analysis underscores the importance of rigorous testing, formal verification, and careful code review in cryptographic software development.

5.2 Mismanagement of Cryptographic State or Context

Mismanagement of cryptographic state or context encompasses bugs where essential flags, parameters, or states are improperly initialized, maintained, or propagated during execution. This includes operation mode flags, arithmetic parameters, and contexts for multi-step processes like key generation or signing. Beyond logical errors, language-specific factors—such as data type sizes, memory management, compiler optimizations, or platform quirks—can exacerbate risks, causing inconsistent state handling across environments.

– Incorrect propagation of flags controlling critical behavior, such as constant-time execution.
– Failure to reset or reinitialize contexts, leading to stale or incorrect data being used in subsequent operations.
– Errors in transferring state between different functions or modules, resulting in loss of important cryptographic properties.

One of the most critical consequences of mismanaged cryptographic state is the failure to maintain constant-time execution. Many cryptographic protocols rely on constant-time operations to prevent attackers from inferring sensitive data through side-channel analysis.

BN_FLG_CONSTTIME Flag Propagation Bug in OpenSSL

Background A notable example of mismanagement of cryptographic state is the `BN_FLG_CONSTTIME` flag propagation bug in OpenSSL. This bug, affected the `BN_MONT_CTX_set` function, a critical component in the RSA key generation process. The `BN_FLG_CONSTTIME` flag is used to enforce constant-time execution during operations involving sensitive data, such as RSA prime numbers p and q. Failure to propagate this flag correctly meant that some operations could inadvertently be executed in variable time, increasing the risk of timing attacks.

Mathematical Context The security of RSA relies on the difficulty of factoring N to recover p and q. However, during the key generation process, if operations involving p and q are not performed in constant time, an attacker could exploit timing variations to infer information about the primes.

Montgomery multiplication is commonly used in RSA implementations to perform modular multiplication efficiently without explicit division. It computes:

$$\text{Mont}(a, b, N, R) = (a \cdot b \cdot R^{-1}) \mod N,$$

where R is a power of 2 larger than N. The `BN_MONT_CTX_set` function is responsible for setting up the Montgomery context used in these operations. Proper propagation of the `BN_FLG_CONSTTIME` flag ensures that all intermediate steps are executed in constant time.

Defect Description The defect in OpenSSL arose because the `BN_FLG_CONSTTIME` flag, set on input parameters p and q, was not correctly propagated to the intermediate `BIGNUM` objects used during the Montgomery setup. As a result, certain intermediate operations could switch to variable-time execution, leaking timing information about p and q.

To illustrate, consider the following simplified example. Let p and q be two 1024-bit primes, and let M be the Montgomery context:

$$M = (p \cdot q) \mod R.$$

If the flag enforcing constant-time multiplication is not correctly propagated, an attacker measuring the time taken for this multiplication could infer properties of p and q, such as their bit length or specific bit patterns.

Impact The failure to maintain constant-time execution during RSA key generation exposes the system to timing attacks. In such an attack, an adversary could repeatedly trigger RSA key generation and measure the time taken for each operation. By analyzing timing variations, the attacker might deduce information about the prime factors p and q, compromising the generated RSA keys.

This vulnerability is particularly concerning in environments where RSA key generation occurs frequently, such as shared cloud platforms or multi-tenant systems. An attacker co-located on the same hardware could exploit the timing side channel to recover private keys generated by other users.

Code Snippet The bug was fixed in OpenSSL by ensuring that the `BN_FLG_CONSTTIME` flag is correctly propagated to all relevant `BIGNUM` objects during Montgomery context setup.

```
if (p->flags & BN_FLG_CONSTTIME)
    ret->flags |= BN_FLG_CONSTTIME;
```

Explicitly checks the `BN_FLG_CONSTTIME` flag on input parameters and propagating it to the result, the fix ensures that all subsequent operations maintain constant-time execution.

Analysis of the Error In Montgomery-based modular multiplication, intermediate results are stored in temporary `BIGNUM` objects that must inherit the `BN_FLG_CONSTTIME` flag. If these objects revert to variable-time operations, partial timing information about the secret prime factors can leak. Propagating `BN_FLG_CONSTTIME` to all intermediate objects, as shown in the fixed code snippet, restores uniform execution time and closes this side channel.

5.3 Incorrect Implementation of Cryptographic Algorithms

Incorrect implementation of cryptographic algorithms refers to a class of bugs that arise from errors in coding cryptographic primitives or operations.

Common types of errors in this category include:

- Incorrect arithmetic operations, such as improper addition, subtraction, or reduction in modular arithmetic.
- Mishandling of cryptographic primitives, leading to incorrect outputs or insecure behavior.
- Failure to adhere to the algorithm's mathematical specifications (group/ring for example), resulting in deviations from the expected cryptographic properties.

Ed25519 Key Handling Bug in OpenSSL

Background A notable example of an incorrect implementation bug is the Ed25519 key handling issue in OpenSSL. This bug affects version 3.0.0-alpha2 and involves cryptographic operations using Ed25519 keys, such as signing and public key extraction. Functions like `X509_sign`, `X509_CRL_sign`, and `X509_REQ_sign` fail to properly process Ed25519 keys, leading to incorrect outputs or failures in signing operations.

Mathematical Context The Ed25519 signature scheme is based on elliptic curve cryptography over the curve defined by the equation (eq:**??**):

$$y^2 = x^3 + 486662x^2 + x \mod 2^{255} - 19.$$

The algorithm involves scalar multiplication on the curve, which is used for both key generation and signature verification.

k is used to generate a signature (R, s) for a message m, where (eq:**??**) with G being the base point of the curve and n the curve's order.

Defect Description The defect in OpenSSL lies in the incorrect handling of Ed25519 public keys during cryptographic operations. Specifically, the `EVP_PKEY_id` function, which is used to identify the key type, sometimes returns 0 instead of the correct identifier for Ed25519 keys (`NID_ED25519`). This incorrect identification causes subsequent operations, such as signing, to fail or return incorrect results.

Additionally, during low-level arithmetic operations involving Ed25519 keys, errors in handling partially reduced values occur. For example, when performing addition or subtraction modulo $2^{255} - 19$, the implementation fails to correctly handle cases where the intermediate result exceeds the modulus.

Impact This bug impacts several cryptographic operations involving Ed25519 keys, particularly in the context of signing X.509 certificates, Certificate Revocation Lists (CRLs), and Certificate Signing Requests (CSRs). Failure to properly identify and process Ed25519 keys can lead to:

– Inability to generate valid digital signatures, resulting in failures in secure communications and certificate management.
– Potential inconsistencies in recognizing and processing Ed25519 keys across different functions, undermining the interoperability of cryptographic systems.
– Compromised integrity of cryptographic operations, affecting the overall trustworthiness of systems using OpenSSL for Ed25519-based signatures.

Code Snippet The following assembly code snippet shows part of the flawed implementation in the handling of addition and subtraction operations for Ed25519 keys:

```
@@ -698,12 +698,16 @@
  add %rax,$acc0
  adc \$0,$acc1
- mov $acc0,8*0(%rdi)
  adc \$0,$acc2
  mov $acc1,8*1(%rdi)
  adc \$0,$acc3
  mov $acc2,8*2(%rdi)
+ sbb %rax,%rax // cf -> mask
  mov $acc3,8*3(%rdi)
+ and \$38,%rax
+ add %rax,$acc0
+ mov $acc0,8*0(%rdi)
  ret
.size x25519_fe64_add,.-x25519_fe64_add
```

Analysis of the Error The provided assembly code snippet illustrates a critical flaw in the handling of addition for Ed25519 finite field elements modulo $p = 2^{255}-19$. The faulty code (lines with a - prefix) performs multi-precision addition and stores the result without checking for overflow.

This omission results in cases where the sum exceeds $2^{255} - 19$ without proper reduction, leading to incorrect results.

In the corrected code (lines with a `+` prefix), the fix introduces a conditional reduction

- Overflow detection: The instruction `sbb %rax, %rax` sets `%rax` to 0 or -1 depending on the carry flag (`cf`), creating a mask.
- Conditional reduction: The mask is multiplied by 38 (derived from $2^{255} - 19 \bmod 2^{64}$) and added to the least significant limb `$acc0`. This ensures that when overflow occurs, the result is correctly reduced modulo $2^{255} - 19$.

This example highlights how a single missing reduction step can result in significant cryptographic failures, emphasizing the importance of correct arithmetic handling in finite fields.

5.4 Missing Input Validation HFBs

Missing input validation refers to a class of bugs where inputs to cryptographic functions are not properly checked for correctness, size, or format before being processed. Input validation is crucial in cryptographic software because improper handling of malformed or unexpected inputs can lead to severe vulnerabilities, such as buffer overflows, denial of service (DoS) attacks, or memory corruption.

Input validation errors often arise due to insufficient checks on:

- Buffer sizes for inputs and outputs, leading to out-of-bounds memory access or buffer overflows.
- Presence of mandatory parameters in cryptographic protocols (e.g., mask generation functions in RSA-PSS).
- Validity of key or public parameter ranges in cryptographic key exchanges.

Failure to implement robust input validation can result in subtle yet severe security flaws. Attackers can exploit these weaknesses by crafting malicious inputs designed to trigger incorrect behavior.

[CVE-2016-2182]: Out-of-Bounds Write in OpenSSL `BN_bn2dec` Function

Background This is a vulnerability in the OpenSSL library affecting the `BN\_bn2dec` function, which converts large integers represented as `BIGNUM` objects to their decimal string representations. This function is commonly used in cryptographic protocols to display or log large integers, such as RSA moduli or Diffie-Hellman parameters.

The vulnerability arises due to inadequate input validation during the conversion process. Specifically, the function fails to properly validate the size of the buffer used to store intermediate results, leading to a potential out-of-bounds write when processing large inputs.

Mathematical Context The `BN\_bn2dec` function operates on `BIGNUM` values, which are arbitrary-precision integers. Given a `BIGNUM` a, the function computes its decimal representation by repeatedly dividing a by a power of 10 until the entire number is converted. The decimal digits are stored in a character buffer, with the result being returned as a null-terminated string:

$$\text{Decimal representation} = \sum_{i=0}^{k-1} d_i \cdot 10^i,$$

where d_i are the decimal digits of the number.

During this process, it is essential to allocate a buffer large enough to hold all decimal digits, including an extra byte for the null terminator. If the buffer size is underestimated, subsequent writes may exceed the allocated memory, resulting in a buffer overflow.

Defect Description The defect occurs in the calculation of the buffer size required to store the decimal representation. The original code allocates the buffer using the following expression:

```
bn_data = OPENSSL_malloc((num / BN_DEC_NUM + 1) * sizeof(
    BN_ULONG));
```

Here, `num` represents an estimate of the number of bits required to represent the decimal value, while `BN_DEC_NUM` is a constant related to the base used for conversion. The allocation logic fails to account for cases where `num` slightly underestimates the actual number of decimal digits needed, leading to insufficient buffer allocation.

The corrected code introduces a new variable `bn_data_num` to properly track the buffer size, ensuring that the allocated memory is always sufficient:

```
bn_data_num = num / BN_DEC_NUM + 1;
bn_data = OPENSSL_malloc(bn_data_num * sizeof(BN_ULONG));
```

Additionally, bounds checks are added during the conversion process to prevent out-of-bounds writes.

Impact The impact of this vulnerability is significant because it allows for out-of-bounds memory writes, which can lead to memory corruption, application crashes, or, in some cases, remote code execution. An attacker could exploit this flaw by providing a specially crafted `BIGNUM` input that triggers the buffer overflow, potentially compromising the security of applications relying on OpenSSL for cryptographic operations.

This vulnerability is particularly concerning in scenarios where untrusted inputs are processed by the `BN_bn2dec` function, such as in public key infrastructures (PKI) or digital certificate validation.

Code Snippet The following code snippet highlights the changes made to fix the vulnerability by introducing proper bounds checks and ensuring correct buffer allocation:

```
@@ -62,6 +62,7 @@ char *BN_bn2dec(const BIGNUM *a)
     char *p;
     BIGNUM *t = NULL;
     BN_ULONG *bn_data = NULL, *lp;
+    int bn_data_num;

     /*-
      * get an upper bound for the length of the decimal
    integer
@@ -71,7 +72,8 @@ char *BN_bn2dec(const BIGNUM *a)
      */
     i = BN_num_bits(a) * 3;
     num = (i / 10 + i / 1000 + 1) + 1;
-    bn_data = OPENSSL_malloc((num / BN_DEC_NUM + 1) * sizeof(
    BN_ULONG));
+    bn_data_num = num / BN_DEC_NUM + 1;
+    bn_data = OPENSSL_malloc(bn_data_num * sizeof(BN_ULONG));
     buf = OPENSSL_malloc(num + 3);
     if ((buf == NULL) || (bn_data == NULL)) {
         BNerr(BN_F_BN_BN2DEC, ERR_R_MALLOC_FAILURE);
@@ -93,7 +95,11 @@ char *BN_bn2dec(const BIGNUM *a)
         i = 0;
         while (!BN_is_zero(t)) {
             *lp = BN_div_word(t, BN_DEC_CONV);
+            if (*lp == (BN_ULONG)-1)
+                goto err;
             lp++;
+            if (lp - bn_data >= bn_data_num)
+                goto err;
         }
         lp--;
```

This example highlights how missing input validation in cryptographic libraries can lead to critical vulnerabilities, such as buffer overflows. Robust input validation is essential in cryptographic software to ensure that inputs are correctly checked for size and format before being processed. The fix introduced in OpenSSL ensures that `BN_bn2dec` properly validates buffer sizes and prevents out-of-bounds writes, thereby mitigating the risk of memory corruption and potential exploitation.

5.5 Misconfiguration or Inconsistent Parameter Handling

Misconfiguration or inconsistent parameter handling refers to a class of bugs where cryptographic parameters are either improperly validated, incorrectly set, or inconsistently handled during cryptographic operations. These issues often arise when parameters such as primes, generators, or exponents are not properly verified or managed, leading to weakened security or incorrect cryptographic outputs.

In cryptographic protocols, parameters play a crucial role in ensuring the correctness and security of operations. Examples include:

- Primes and generators in Diffie-Hellman and ElGamal key exchanges.
- Curve parameters in elliptic curve cryptography (ECC).
- Initialization vectors (IVs) and nonces in symmetric encryption schemes.

Failure to correctly configure or consistently handle these parameters can introduce subtle vulnerabilities. Attackers may exploit improperly validated parameters to launch various attacks, such as key recovery, plaintext recovery, or subgroup confinement attacks, ultimately compromising the cryptographic protocol.

[pycrypto#985164]: Generator Validation in ElGamal Encryption Scheme

Background This bug affects the PyCrypto library's implementation of the ElGamal encryption scheme, which is a widely used public-key encryption method based on the Diffie-Hellman key exchange principle. The security of ElGamal encryption relies on the correct selection of parameters, including the generator g, which must belong to a secure subgroup of a large cyclic group defined by a prime p.

The generator g is used to derive ephemeral keys during encryption. To ensure security, g must have a sufficiently large order, meaning it must generate a large subgroup of the group Z_p^*. If the generator does not have the correct order, it can lead to weakened encryption, allowing attackers to exploit smaller subgroups to recover partial or complete information about the encrypted message.

Mathematical Context The ElGamal encryption scheme operates in the multiplicative group Z_p^*, where p is a large prime, and the generator g is an element of this group. Given a message m, ElGamal encryption involves the following steps:

1. Select a random ephemeral exponent k such that $1 \leq k \leq p - 2$.
2. Compute the shared value:
$$C_1 = g^k \mod p.$$
3. Compute the masked message:
$$C_2 = m \cdot y^k \mod p,$$
where $y = g^x \mod p$ is the public key, and x is the private key.

For security, it is essential that g has a large order, ideally dividing $p - 1$, the order of the group Z_p^*. If g has a small order, the shared value C_1 may only take a limited number of values, making it possible for an attacker to perform a brute-force attack on the encrypted message by testing all possible values of k.

Defect Description The defect in PyCrypto's implementation arises because the generator g is not properly validated to ensure it belongs to a secure subgroup of Z_p^*. Specifically, the code responsible for validating the generator contains an incomplete check, allowing insecure generators to be used in the encryption process. The problematic code is as follows:

```
if safe and pow(obj.g, q, obj.p) == 1:
    safe = 0
```

Here, q is a factor of $p-1$. The check verifies whether $g^q \equiv 1 \pmod{p}$, which would indicate that g generates a subgroup of order q. However, this check alone is insufficient because it does not ensure that g generates the entire group or a sufficiently large subgroup.

The corrected implementation involves ensuring that g has a large prime order and does not belong to a small, insecure subgroup. The fix introduced an additional validation step to verify that g has the correct order by checking that g does not satisfy $g^d \equiv 1 \pmod{p}$ for any small divisor d of $p-1$.

Impact The impact of this bug is significant because it weakens the security of the ElGamal encryption scheme. By allowing insecure generators to be used, the implementation becomes vulnerable to subgroup confinement attacks, where an attacker forces the encryption process to use a small subgroup, making it feasible to recover the ephemeral key k through brute-force enumeration.

In practice, this vulnerability could lead to the exposure of sensitive data encrypted using the PyCrypto library. Attackers could exploit the improper generator validation to compromise encrypted communications, especially in scenarios where ElGamal is used for key exchange or data encryption in distributed systems.

Code Snippet The following code snippet shows the corrected implementation, which ensures proper generator validation by incorporating additional checks for the order of the generator:

```
if safe:
    for d in divisors(p - 1):
        if pow(obj.g, d, obj.p) == 1:
            raise ValueError("Invalid generator: small order
   detected")
```

This example illustrates how misconfiguration or inconsistent parameter handling can lead to critical vulnerabilities in cryptographic protocols. Proper validation of parameters, such as generators in ElGamal encryption, is essential to maintaining the security of cryptographic operations. The fix introduced in PyCrypto ensures that only secure generators with sufficiently large orders are used, mitigating the risk of subgroup confinement attacks and ensuring the robustness of the encryption scheme.

5.6 Failure to Ensure Constant-Time Operations

Failure to ensure constant-time operations refers to a class of vulnerabilities where cryptographic functions exhibit timing variations based on input values or intermediate states. Such timing variations can be exploited by attackers through side-channel attacks, which aim to extract sensitive information by observing the time taken to perform cryptographic operations. This category of bugs is particularly concerning in environments where attackers have physical or remote access to monitor timing information, such as cloud infrastructures or virtualized systems.

Cryptographic algorithms are often designed under the assumption that their operations are performed in constant time, meaning that the time required for execution should not depend on the secret. However, improper implementations may inadvertently introduce timing discrepancies, leading to vulnerabilities. Common causes include:

– Conditional branches that depend on secret data.
– Non-constant time arithmetic operations, such as modular exponentiation.
– Cache access patterns that vary based on secret information.

Failure to maintain constant-time behavior can enable attackers to deduce private keys, nonces, or other sensitive cryptographic material through timing analysis. Such attacks have been demonstrated on a wide range of cryptographic algorithms, including RSA, DSA, and ECC.

[CVE-2018-0737]: Cache Timing Side-Channel Attack in RSA Key Generation

Background During the key generation process, several modular exponentiation operations are performed to generate the prime candidates. If these operations are not performed in constant time, timing variations may occur that can leak information about the prime candidates. An attacker who can observe these timing variations may be able to recover the prime numbers, compromising the private key.

Mathematical Context RSA key generation involves (section:**??**). To generate p and q, OpenSSL uses a probabilistic primality testing algorithm, which involves computing modular exponentiations of the form:

$$a^d \mod N,$$

where d is a candidate exponent derived from the prime factorization process.

In a timing side-channel attack, the attacker measures the time taken by the modular exponentiation operation. If the operation exhibits timing variations based on the value of d, the attacker can infer information about d by observing multiple such operations. This information can eventually be used to deduce the prime numbers p and q.

Defect Description The defect in OpenSSL lies in the modular exponentiation routine used during RSA key generation. The routine includes conditional branches and arithmetic operations that depend on the value of the exponent, leading to timing variations. Specifically, cache access patterns during the computation vary based on the value of the exponent, making the routine susceptible to cache timing attacks.

The following code snippet highlights the relevant section where constant-time behavior is enforced by setting a constant-time flag for the prime numbers:

```
@@ -157,6 +157,7 @@ static int rsa_builtin_keygen(RSA *rsa, int bits, int primes, BIGNUM *e_value,
              pinfo = sk_RSA_PRIME_INFO_value(prime_infos, i - 2);
              prime = pinfo->r;
          }
+         BN_set_flags(prime, BN_FLG_CONSTTIME);

          for (;;) {
 redo:
```

The added `BN_set_flags` function call ensures that the modular exponentiation is performed using constant-time operations, thereby mitigating timing variations that could lead to side-channel attacks.

Impact The impact of this vulnerability is significant, particularly in shared environments such as cloud infrastructures or virtualized systems where attackers can monitor cache timings. By exploiting this vulnerability, an attacker could potentially recover the private RSA key during the key generation process, compromising the security of encrypted communications or digital signatures.

This example demonstrates how failure to ensure constant-time operations can lead to critical vulnerabilities in cryptographic systems. The fix introduced in OpenSSL mitigates the risk by enforcing constant-time behavior during RSA key generation. Cryptographic libraries must ensure that all operations, especially those involving secret data, are performed in constant time to prevent timing side-channel attacks.

5.7 Data Collection and Vectors

Collecting, categorizing, and analyzing the diverse bugs discovered in cryptographic software provides valuable insights into the pitfalls of real-world implementations. This compilation process aids automated testing, static analysis, and formal verification efforts by furnishing concrete examples of what can go wrong, as well as why and how these errors arise. Beyond serving as a cautionary guide, this corpus of bugs also offers an opportunity to train Large Language Models (LLMs) to automatically detect, generate new HFBs and generate test vectors (Known Answer Tests) to better secure cryptographic libraries.

For each bug, a detailed vector was compiled to aid in the understanding and categorization of its impact. The vector contains the following categories and more:

1. ID: The unique identifier for the bug, such as a CVE (Common Vulnerabilities and Exposures) number or a specific project-related identifier (commit).
2. BigType: A high-level categorization of the bug, such as "Carry" or "Special", describing its class.
3. Language: The programming language where the bug was identified, e.g., Assembly or C.
4. SubType: A more granular categorization of the bug, such as "Montgomery Sq" for issues related to Montgomery squaring.
5. Impact: A description of the bug's effect, such as undermining operations, side-channel vulnerabilities, or incorrect results.
6. DangerLvl: A numerical score indicating the severity of the bug, on scale 0.0–10.0. based on CVE danger scores.

Improving Data Collection and Testing The full catalog contains 50+ documented HFBs their ananysis and compilation underscores a few overarching lessons:

1. Assembly Complexity: A disproportionately large share of bugs occurs in hand-rolled assembly, where readability and maintainability are often sacrificed for optimization.
2. Carry Propagation Prevalence: Carry-related pitfalls rank among the most common, highlighting how multi-precision arithmetic operations can become error-prone when optimizations or incomplete checks are introduced.
3. Cross-Platform Inconsistencies: Slight differences in compiler behavior, CPU feature sets (BMI2, ADX), or endianness can trigger bugs that are otherwise invisible in a single test environment.

Equipped with these vectors, researchers and developers can leverage the corpus to build fuzzers and static analyzers that specifically target known hard to find cryptographic pitfalls. Large language models could also ingest these bug descriptions to better identify and classify similar issues in novel codebases or to generate synthetic test cases, or new undetected bugs that reproduce the exact conditions under which a bug emerges. In this way, the systematic study and documentation of HFBs benefits the broader security community by preventing regressions and future oversights.

6 Conclusion

In this survey paper, we have delivered a thorough examination of Hard-to-Find Bugs (HFBs) in cryptographic software, with a specific emphasis on public-key cryptography. These subtle bugs present unique challenges to the security and reliability of cryptographic systems due to their low probability of occurrence, high complexity, and resistance to detection by conventional testing methods. Our analysis underscores the critical need to address these elusive vulnerabilities to safeguard the integrity of digital security.

We systematically evaluated a spectrum of testing methodologies tailored to uncover HFBs, including differential testing, static analysis, Monte Carlo testing, fuzzing, formal verification, and Known Answer Tests (KATs). Each method offers distinct advantages: differential testing excels at detecting inconsistencies across implementations; static analysis provides rigorous logical error detection and ensures properties like constant-time execution; Monte Carlo testing offers statistical input coverage; fuzzing reveals robustness issues under malformed inputs; formal verification delivers mathematical assurances of correctness; and KATs, exemplified by tools like Wycheproof, provide essential baseline validation by testing against known vectors and edge cases. Wycheproof, in particular, enhances KATs by targeting known HFBs, as demonstrated in our accompanying C implementation for OpenSSL, bridging practical testing with real-world applicability.

To advance HFB detection, we proposed a layered testing framework that integrates KATs, differential testing, and continuous fuzzing. This pragmatic approach combines deterministic, comparative, and dynamic testing strengths, empowering developers to bolster the security of their cryptographic implementations effectively. However this is still not enough for critical software which should also be mathematically guaranteed though formal verification.

A key contribution of this survey is the structured collection of real-world HFBs, meticulously categorized in Section 5 to illuminate their diverse nature and aid in mitigation efforts. This bug collection serves as a vital resource for the cryptographic community, offering concrete examples that highlight common pitfalls and reinforce the necessity of specialized testing techniques. By documenting these bugs—ranging from carry propagation errors to constant-time failures—we provide a foundation for improved testing strategies and prevention of recurring vulnerabilities.

6.1 Key Takeaways

Our analysis yields several critical insights:

- Multi-Method Necessity: No single testing methodology suffices to detect all HFBs; a combination of approaches is essential for comprehensive coverage.
- Formal Verification's Role: Mathematical proofs offer unparalleled assurance, particularly for high-stakes cryptographic implementations, complementing empirical methods.
- Value of Bug Collections: Structured catalogs of HFBs inform better testing practices, enabling developers to anticipate and mitigate elusive vulnerabilities proactively.

6.2 Future Directions

The persistent challenge of HFBs demands a continuous and adaptive security approach. Looking forward, we advocate for further research into HFBs within emerging domains, such as post-quantum cryptography. As new algorithms introduce novel bug classes, tailored testing methodologies may be required to ensure their robustness. The difficulty is also highlighted by CISA, NSA and NIST not recommending hybrid classical and post-quantum cryptographic implementations due to their complexity [13]. Additionally, leveraging advanced techniques like machine learning—trained on bug collections to predict or generate new HFBs—could enhance detection capabilities and push the boundaries of cryptographic security.

In conclusion, the methodologies, framework, and insights presented in this survey equip developers and researchers with robust tools to confront HFBs head-on. By integrating rigorous testing with a deep understanding of HFB characteristics, we can advance toward more resilient and trustworthy cryptographic implementations, fortifying the bedrock of digital security in an ever-evolving threat landscape.

References

1. Almeida, J.B., Barbosa, M., Barthe, e.a.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823. ACM (2017). https://doi.org/10.1145/3133956.3134078
2. for anonimity, H.: Wycheproof-c: A c cryptographic test suite. https://anonymous.4open.science/r/wycheproof-c-3C6C/README.md (2025)
3. Barthe, G., Grégoire, B., Strub, P.Y.: Formal certification of cryptographic security proofs: Fully automated verification of computational proofs. Journal of Cryptology **32**(3), 703–741 (2019)
4. Barthe, G., Grégoire, B., Zanella-Béguelin, S.: Formal certification of code-based cryptographic proofs. In: IEEE Computer Security Foundations Symposium (CSFW). pp. 246–260 (2005)
5. Bentley, J.A., Flanagan, C.G., Bhargavan, K., Zanella-Béguelin, S.: Verified cryptographic implementations: Proving security properties of cryptographic code. In: Automated Deduction - CADE. pp. 36–52 (2020)
6. Beringer, L., Petcher, A.: Verified cryptographic code for everybody. In: IEEE Symposium on Security and Privacy. pp. 1175–1191 (2017)
7. Birr-Pixton, J., et al.: rustls: A modern tls library in rust. https://github.com/rustls/rustls (2023), accessed: 2023-10-01
8. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). pp. 417–431. IEEE (2008). https://doi.org/10.1109/SP.2008.30
9. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: International conference on the theory and applications of cryptographic techniques. pp. 37–51. Springer (1997)
10. Bressana, P., Zilberman, N., Soulé, R.: Finding hard-to-find data plane bugs with a pta. In: Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies. pp. 218–231 (2020)

11. Chen, Y., Su, Z.: Differential testing for cryptographic libraries. In: USENIX Security Symposium. pp. 1107–1122 (2016)
12. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., et al.: Continuous formal verification of amazon s2n. pp. 430–446. Springer (2018)
13. CISA, NSA, NIST: Quantum-readiness: Migration to post-quantum cryptography. https://media.defense.gov/2022/Sep/07/2003071836/-1/-1/0/CSI_CNSA_2.0_FAQ_.PDF (2022)
14. De Ruiter, J., Poll, E.: Protocol state fuzzing of {TLS} implementations. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 193–206 (2015)
15. Denis, F.: libsodium - a modern, portable, easy to use crypto library (2013), https://libsodium.org, version 1.0.18, Accessed: Mar. 3, 2025
16. fail0verflow: Console Hacking 2010: PS3 Epic Fail. Presented at the 27th Chaos Communication Congress (27C3) (December 2010), https://fail0verflow.com/media/27c3_console_hacking_2010_ps3.pdf
17. Go Team: Go cryptography libraries. https://pkg.go.dev/crypto (2023), accessed: 2023-10-01
18. Hax Team: Hax: A rust verification toolchain for security-critical software (2023), available at https://github.com/hax-rust/hax
19. Hwang, V.: Formal verification of emulated floating-point arithmetic in falcon. In: International Workshop on Security. pp. 125–141. Springer (2024)
20. Isinger, I.L., et al.: curve25519-dalek: A pure-rust implementation of curve25519 and related cryptographic primitives. https://github.com/dalek-cryptography/curve25519-dalek (2018)
21. Jero, S., Pacheco, M.L., Goldwasser, D., Nita-Rotaru, C.: Leveraging textual specifications for grammar-based fuzzing of network protocols. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 9478–9483 (2019)
22. Joye, M., Tunstall, M., et al.: Fault analysis in cryptography, vol. 147. Springer (2012)
23. Kasak, D.: Rust vs. c: A performance comparison in systems programming (2018), blog post, accessed at https://deniskasak.github.io/rust-vs-c-perf
24. Libcrux Team: Libcrux: A formally verified cryptographic library for rust (2023), available at https://github.com/cryspen/libcrux
25. Markelova, A.V.: Vulnerability of rsa algorithm. In: CEUR Workshop Proceedings. vol. 2081, pp. 74–78 (2017)
26. (NSA), N.S.A.: Software memory safety: Mitigating memory safety issues. Tech. rep., National Security Agency (November 2022), https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF
27. Rukhin, A., Soto, J., Nechvatal, e.a.: A statistical test suite for random and pseudorandom number generators for cryptographic applications, vol. 22. US Department of Commerce, Technology Administration, National Institute of ... (2001)
28. Seaborn, T.: Performance analysis of rustcrypto: Aes implementations in rust vs. c (2019), available at https://rustcrypto.org/performance
29. Seals, T.: Attacker social-engineered backdoor code into xz utils. https://www.darkreading.com/attacks-breaches/attacker-social-engineered-backdoor-code-into-xz-utils (2024)
30. Smith, B.: ring: Safe, fast, small crypto using rust. https://github.com/briansmith/ring (2023), accessed: 2023-10-01
31. team, G.T.: wycheproof. https://github.com/C2SP/wycheproof (2018-Now)

32. Team, N.G.R.: On multiplications with unsaturated limbs (2023), `https://www.nccgroup.com/us/research-blog/on-multiplications-with-unsaturated-limbs/`
33. Team, R.: Rustcrypto: Cryptographic algorithms implemented in pure rust. `https://github.com/RustCrypto` (2023), accessed: 2023-10-01
34. Zalewski, M.: American fuzzy lop. `https://github.com/google/AFL` (2014)
35. Zalewski, M.: Oss-fuzz: Five years later. `https://google.github.io/oss-fuzz/` (2020)