Hard-to-Find Bugs in Public-Key Cryptographic Software: Classification and Test Methodologies

Matteo Steinbach, Johann Großschädl, and Peter B. Rønne

DCS and SnT, University of Luxembourg,
6, avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
matteo.steinbach.pro@gmail.com
{johann.groszschaedl,peter.roenne}@uni.lu

Abstract. Programming bugs and flaws can have fatal consequences for the security of cryptographic software and may allow an attacker to bypass authentication, forge signatures, decrypt sensitive data, or even completely reveal secret keys. Certain categories of bugs, such as subtle carry-propagation flaws in large-integer or prime-field arithmetic carried out by many public-key cryptosystems, manifest only under very specific and, therefore, extremely rare input conditions, which makes them hard to detect with conventional software testing methodologies. While there exist a few papers that describe such Hard-to-Find Bugs (HFBs) and study their security implications, a more comprehensive treatment and systematization are still lacking. The present paper aims to fill this gap and analyzes the challenges posed by HFBs in software implementations of public-key cryptosystems. More concretely, we define and categorize HFBs, provide a survey of HFBs that have been found in widely-used open-source cryptography libraries (some of which remained undetected for up to 10 years), and discuss the benefits and limitations of common testing and prevention techniques, including differential testing, static analysis, fuzzing, formal verification, and Known Answer Tests (KATs) tailored to HFBs. Raising awareness of HFBs is important for software developers and security auditors who implement and test cryptographic algorithms for mission-critical systems where correctness and robustness are paramount. By shedding light on subtle implementation flaws and how to reduce their occurrence, this paper contributes to improving the real-world security of public-key cryptosystems.

1 Introduction

Cryptographic algorithms are foundational to secure communication over the Internet, safeguarding the confidentiality, authenticity, and integrity of sensitive data. While many of the widely-used cryptosystems stand on a solid theoretical foundation and have been scrutinized for many years, their security in practice is intrinsically related to the correctness of the implementation [22]. Programming errors and resultant defects (i.e., "bugs") can be found in any non-trivial software, and cryptographic software is certainly no exception [7]. According to McConnell [24, Sect. 22], the "industry average experience is about 1-25 errors per 1000 lines of code for delivered software." Since a software implementation of a single cryptosystem, such as ECDSA, can consist of several hundred Lines of Code (LoC), it is not surprising that a substantial number of bugs have been found in cryptographic libraries [7]. The impact of a bug or flaw in software, in general, and cryptographic software, in particular, can vary significantly. While some bugs are benign and do not cause serious harm, others have catastrophic effects and can even be responsible for a loss of life [34]. More specifically, when it comes to cryptographic software, the immediate consequences of bugs/flaws that slip through quality assurance (e.g., code auditing, testing) and end up in production usually include the incorrect execution of a cryptosystem for certain (potentially very rare) inputs and the leakage of sensitive information via side channels. Indirect (i.e., further) consequences of the former can range from the bypassing of authentication in TLS (e.g., Apple's "`goto fail`" bug [22]) to the forging of signatures (e.g., by exploiting a padding bug that breaks the collision resistance of the signature's hash function [25]) to the decryption of sensitive data (e.g., by taking advantage of a modular reduction bug to leak the private ECDH key of a TLS server [10]). In the worst case, a single bug can enable an attacker to fully recover a secret key with little effort [14]. However, even if an implementation of a cryptosystem is 100% functionally correct (i.e., produces always a correct result if the input is valid and an error code otherwise), it can still leak sensitive information, e.g., through small input-dependent variations in execution time that could be exploited by a timing attack.

Two commonly-used techniques to discover programming errors and defects are source-code reviewing (including auditing by external experts) and software testing (resp., fuzzing). However, both are particularly challenging for cryptographic software. To maximize efficiency, the performance-critical components of cryptographic algorithms are usually written in Assembly language, which is error-prone not only for the developers but also for reviewers and auditors. In addition, the need for resistance against timing-based side-channel attacks adds an extra layer of complexity, as it prohibits secret-dependent memory accesses and conditional branches, respectively [21]. Furthermore, when a cryptographic library aims to support different processor architectures, a number of separate Assembly implementations have to be reviewed/audited, requiring intricate and rare expertise. Cryptographic software also has a number of unique properties with respect to testing or fuzzing that distinguish it from software in other domains. Namely, certain classes of cryptographic software defects, such as subtle carry-handling flaws [7], emerge only under highly specific and extremely rare input conditions. A good example is the carry-propagation bug in OpenSSL's Karatsuba-based long-integer squaring studied by Weinmann [32]. This bug is triggered with a probability of 2^{-64} on affected 32-bit platforms and 2^{-128} on 64-bit processors, respectively. While conventional testing techniques are vital for basic correctness and regression checks, their odds of finding such an elusive bug are practically zero. Thus, it is not surprising that this carry-propagation bug was present in the OpenSSL code-base for 10 years [32], making it a prime example of a *Hard-to-Find Bug* (abbreviated in the following as HFB).

The main goal of this paper is to raise awareness of the existence of HFBs and their implications among cryptographers on the one hand and software developers on the other, whereby we focus on public-key cryptographic algorithms like RSA and Elliptic Curve Cryptography (ECC). These cryptosystems, due to their reliance on costly mathematical operations (e.g., long-integer arithmetic in RSA and finite-field computations in ECC), are especially susceptible to subtle functional bugs and side-channel flaws. Such defects are a natural consequence of inherent algorithmic complexity, sophisticated optimization, platform-related constraints, and (micro-)architectural nuances of the target processor. Software developers are often unaware of the specifics of cryptosystems and the aspects that make cryptographic software unique. For example, even when a software implementation of a public-key cryptosystem is fully functionally correct, it can still contain an HFB and leak secret data, e.g., via some side channel or due to bad randomness [22]. On the other hand, cryptographers often lack knowledge of the complete spectrum of software testing and formal verification techniques that can be used to tackle HFBs. While each of them has its own benefits and shortcomings, the combination of testing (fuzzing) and formal verification can reduce the number of HFBs that slip through and end up in production.

This paper provides a definition of HFBs, describes different characteristics they share, and exemplifies these characteristics with a famous HFB that made headlines world-wide, namely the Sony PlayStation 3 (PS3) bug [14]. We also discuss the benefits and limitations of state-of-the-art software testing methods (both generally and specifically in the context of HFBs), including differential testing, static analysis, Monte Carlo testing, fuzzing, and Known Answer Tests (KATs) tailored to the rare corner cases where HFBs commonly hide. Beyond "detection and elimination," an alternative way to mitigate HFBs is prevention through formal verification of correctness [2,8]. We survey tools and techniques for the formal verification of cryptographic implementations and highlight the advantages of modern programming languages for building high-assurance software. Another contribution of this paper is a collection of HFBs that have been discovered in various open-source projects and their analysis and categorization into six main classes of bugs: carry propagation flaws, mismanagement of state or context, other implementation issues, incomplete input validation, erroneous parameterization, and vulnerability to timing attacks. A more detailed version of this bug collection, which covers 53 HFBs in total and also includes a small source-code excerpt where each bug manifests, is available on GitHub [30]. The GitHub repository also contains an extended 35-page version of this paper.

2 Hard-To-Find Bugs (HFBs)

In the context of software quality assurance, the term *error* generally refers to a human mistake that produces an unintended or incorrect result. Errors can occur in any phase of the development process, e.g., specification, design, implementation, testing, and maintenance. A *defect* is a deficiency or imperfection in a software artifact that prevents it from satisfying the (intended) requirements or specification [24]. Hence, a defect is the consequence of an error, e.g., a false specification due to a misinterpretation of a requirement or a flaw in the design of the software. A *bug* is the manifestation of a defect and typically discovered during the testing or operation of software artifacts. Bugs can affect any phase of the software development process, from specification to maintenance, but we focus primarily on implementation bugs in this paper. Having established the basic terminology, we define a Hard-to-Find Bug (HFB) as follows.

> An HFB is any kind of imperfection of a delivered cryptographic software implementation that can potentially lead to a security vulnerability and remains undetected by state-of-the-art testing techniques.

The word *delivered* emphasizes that HFBs are bugs in official software versions or releases that made it to production systems, i.e., we explicitly exclude bugs in alpha or beta software. Alpha/beta versions often come with critical bugs and are intended for testers and early adopters to identify these issues.

An important category of HFBs consists of those that cause cryptographic software to produce an incorrect result for at least one input combination. Such input combinations are typically extremely rare, making these bugs difficult to uncover through conventional testing or fuzzing. Even worse, incorrect outputs of a cryptosystem can leak sensitive data (e.g., a secret key) to an attacker, as was shown in [10]. However, not all HFBs manifest as incorrect outputs. Some arise from low-level memory management issues, such as dynamically allocated memory (using, e.g., `mallloc` in C) that is never freed. Although these bugs do not affect functional correctness, they can still pose a security risk if sensitive data, e.g., a secret key or a temporary value related to a secret key, remains in memory and becomes accessible to an attacker[1]. Other HFBs stem from flaws in (pseudo-)randomness generation, often referred to as "bad randomness." The security of many cryptosystems depends crucially on random numbers that are unpredictable and uniformly distributed within a specified range; defects in the generation process can have fatal consequences yet are notoriously difficult to detect through standard testing. Timing vulnerabilities represent another class of HFBs: even when outputs are fully correct, variations in execution time can leak information about the secret key via side-channel attacks. These examples illustrate that HFBs encompass a broad range of imperfections, including some that do not affect functional correctness but still can undermine security.

Characteristics of HFBs. Most of the HFBs described in the literature and in bug reports, change logs, mailing lists, and discussion forums of open-source projects share one or more of the following properties.

1. Low Probability of Occurrence: An HFB manifests only under extremely rare and highly specific conditions, such as unique input combinations, special sequences of operations, or particular hardware or timing factors.

[1] Note that our HFB definition does not imply the existence of an exploit; it suffices when the bug has the potential to compromise security.

2. High Complexity and Subtlety: An HFB often results from complex system interactions, subtle hardware behavior, or some intricate algorithmic detail (e.g., improper handling of edge cases).
3. Difficulty in Detection and Reproduction: Conventional testing techniques struggle to identify HFBs, 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 system stack.

Subtle bugs that are hard to find do not only plague cryptographic software but represent a massive challenge across essentially all segments of the software industry. For example, Bressana et al. [9] discuss the difficulty of finding subtle data plane bugs in networking hardware using their Portable Test Architecture (PTA), which revealed hidden issues like throughput degradation under specific traffic conditions. Although their research is on network devices, the discussed challenges closely mirror those in cryptography: both fields are plagued by bugs that only manifest under rare conditions, evading traditional testing methods.

2.1 HFBs in Public-Key Cryptography

Public-key cryptography involves costly low-level arithmetic operations, such as exponentiation in a multiplicative group with an order of a few thousand bits (e.g., RSA, Diffie-Hellman) or scalar multiplication in an additive group of an order of a few hundred bits (ECC schemes, e.g., ECDH, ECDSA). The latter, in turn, requires operations in a finite field, typically a prime field $\mathbb{F}_p$, where p is chosen to enable fast modular reduction. When implemented in software, these operations are performed on multi-precision integers, which are arrays of words or limbs whose length is determined by the wordsize of the target platform. As discussed in the last section, public-key cryptosystems are prone to many kinds of HFBs, including arithmetic defects (e.g., carry propagation issues, overflows of words/limbs, sign handling lapses), flaws in the generation of pseudo-random numbers, and vulnerabilities to side-channel attacks. The public nature of these cryptosystems, allowing adversaries to craft malicious inputs and probe for bugs without having privileged access, increases the risk that HFBs can be exploited to compromise security. For example, in (EC)DH key exchange, a private key is combined with a public key that may be maliciously crafted by an attacker to trigger leakage of (parts of) the private key via an HFB [10].

Example: Sony PS3 Hack. A well-known example for an HFB is the bug in ECDSA signature generation that affected the Sony PlayStation 3 (PS3) game console and opened doors to widespread software piracy [14]. This bug allowed an attacker to compute Sony's private code-signing key from publicly available signatures, thereby completely breaking the chain of trust of the PS3. With the private key exposed, it was possible to sign arbitrary code as if it were official firmware, which made it relatively easy to create custom firmware and execute unauthorized software, such as illegal copies of PS3 games.

Before explaining the bug and its implications in more detail, we first recap on how ECDSA generates and verifies a signature for a given message m. From a mathematical point of view, ECDSA operates in an elliptic curve (sub-)group of prime order n with generator G. The signer has a key-pair (d, Q) where d is an integer in the range of $[1, n-1]$ and $Q = d \cdot G$ is a point on the curve. To obtain a signature for m, the signer computes the hash $h = H(m)$ and chooses a random nonce $k \in [1, n-1]$. Then, the signer performs a scalar multiplication to get a public nonce $R = k \cdot G$, extracts the x-coordinate r_x of R, and derives $s = k^{-1}(h + d\,r_x) \bmod n$. The signature of m is the pair (r_x, s). To verify this signature, one has to hash m and use the public key Q to check if r_x matches the x-coordinate of the point $T = u_1 \cdot G + u_2 \cdot Q$, where $u_1 = h\,s^{-1} \bmod n$ and $u_2 = r\,s^{-1} \bmod n$. When several or many messages are signed using the same private key d, the security of the ECDSA signature scheme critically hinges on the nonces being unique, unpredictable, and uniformly distributed in the range $[1, n-1]$. Accidental nonce re-use can leak the secret key d.

Suppose two signatures are generated using the same nonce k, one for message m_1 with hash h_1, and the other for m_2 with hash h_2. In this case, the two signatures have the same r_x coordinate, but different s values: s_1 and s_2. The difference $s_1 - s_2 = k^{-1}(h_1 + d\,r_x) - k^{-1}(h_2 + d\,r_x) = k^{-1}(h_1 - h_2) \bmod n$ is sufficient to get $k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n$ and, then, recover the private key from either signature by computing, e.g., $d = (s_1\,k - h_1)\,r_x^{-1} \bmod n$. Sony's flawed ECDSA implementation used a fixed nonce k (or insufficiently random nonces), producing signatures with the same r_x for different messages. Hackers collected these signatures and recovered the private key d, which enabled them to sign unauthorized firmware and bypass all security controls.

Cryptanalytic attacks enabled by "bad randomness" (including a complete lack of randomness) have a long history in public-key cryptography and broke the security of many real-world systems. Possible causes for randomness errors range from a lack of implementer awareness to defects in an operating-system or hardware component that is involved in the generation of (pseudo-)random numbers. Modern cryptographic libraries come with their own Pseudo-Random Number Generator (PRNG), which is initially seeded, and then occasionally re-seeded, with "true" (in terms of non-deterministic) randomness collected by the operating system from hardware events, interrupts, or special CPU instructions like `rdseed`. Most standalone ECDSA implementations, on the other hand, use randomness provided by the operating system's PRNG (e.g., `/dev/urandom` in Linux) to avoid the hassle of maintaining a PRNG state. Both approaches can succumb to implementation errors. For example, a bug in the OpenSSL crypto library of Debian Linux distributions from 2008 (CVE-2008-0166) crippled the seeding of OpenSSL's PRNG, making it predictable and causing many of the generated cryptographic keys to be weak. Commenting out two lines of source code reduced the entropy of the seed to the process-ID, i.e., only 15 bits. More recently, it became known that AMD Zen5 CPUs are affected by a flaw in the `rdseed` instruction, which is used by many operating systems as one of several sources of entropy. Under certain conditions, `rdseed` returns the value 0 more often than true randomness would allow and still signals success by setting the carry flag to 1 (CVE-2025-62626). A different kind of mistake, often committed by inexperienced developers, is to use a PRNG that is not suitable for cryptographic purposes. A typical example in the context of the Java language is the generation of ECDSA nonces with the `java.util.Random` class instead of the `java.security.SecureRandom` class. The former employs a non-cryptographic PRNG based on a linear congruential formula with a period of 2^{48}.

Overall, the ECDSA bug in Sony's PS3, and similar bugs due to "bad randomness," aligns well with the characterization of HFBs outlined earlier:

1. Low Probability of Occurrence: Depending on the defect in the generation of (pseudo-)random numbers, a nonce re-use may happen only after a large amount of signing operations (e.g., 2^{48} when `java.util.Random` is used as PRNG to obtain nonces).
2. High Complexity and Subtlety: The nonces for ECDSA signatures have to be unique, unpredictable, and uniformly distributed. Even small deviations may be exploitable and facilitate key recovery [3], making nonce generation a challenging task. In addition, from a purely algorithmic point of view, an ECDSA signature generated with a re-used nonce is still a correct signature (in the sense that the verification yields the correct result).
3. Difficulty in Detection and Reproduction: Conventional testing frameworks for ECDSA check for functional correctness, but are (normally) not able to detect accidental nonce misuse caused by PRNG flaws. While an extension of testing frameworks to catch duplicated nonces is certainly possible, the probability of finding a duplicate can be extremely low and depends on the properties of the defect in (pseudo-)random number generation.
4. Cross-Disciplinary Nature: The generation of (pseudo-)random nonces is an intricate problem as it requires expertise not only in cryptography, but also in operating systems (e.g., entropy collection, reseeding strategies) and even hardware (e.g., failure modes of entropy sources). This task is further complicated by the fact that many relevant implementation details, such as the post-processing/whitening of entropy, are often poorly documented or even completely opaque, especially in closed-source operating systems.

2.2 Exploitability and Impact

The exploitability of an HFB depends less on how often (or rarely) it manifests under benign conditions and more on whether an adversary is able to drive the execution into the HFB's "triggering region" and observe a usable output. In public-key cryptography, attackers often have direct control over certain inputs (e.g., chosen messages, public keys, input encoding), which enables systematic exploration of edge cases that are practically unreachable by arbitrary/random inputs. Equally important is observability: many HFBs become exploitable in practice only when their outputs/effects are externally visible (e.g., in the form of signatures, accept/reject decisions, detailed success/error codes, measurable timing differences, protocol abortions, system crashes). Finally, exploitability is amplified by repeatability; if a bug can be triggered in a deterministic fashion or with a certain probability under repeated trials, an attacker can accumulate evidence and/or reduce uncertainty through statistics. We propose to assess the exploitability of an HFB by considering the following aspects.

1. Adversarial control: Can an attacker directly choose or otherwise influence inputs and/or parameters? Is the execution environment remote or local?
2. Signal strength: Does the HFB produce a clean, low-noise "signal" (e.g., an incorrect result that is public) or only a weak side channel requiring a large number of samples (e.g., a minimal timing bias)?
3. Trigger reliability: Is the HFB's triggering condition deterministic, stateful (i.e., requiring a sequence), or purely probabilistic?
4. Trial or query effort and post-processing complexity: What is the expected number of trials/queries to get the desired "signal" (online complexity) and how costly is the post-processing, e.g., to extract a secret key from a given set of noisy timing measurements (offline complexity)?

As outlined before, an HFB can be simultaneously rare in normal operation yet highly exploitable in adversarial settings. The impact of an exploit can be classified according to the severity of the resulting vulnerability, which enables developers to prioritize fixes based on potential damage. Such a classification is beneficial for a better understanding of the importance and urgency of actions for remedy. Aspects to consider for impact classification include:

1. Security implications, roughly ordered by severity: (a) full key compromise (log-term key vs. short-term key, authentication key vs. encryption key), (b) forgeability or authentication failure, (c) loss of confidentiality without full key compromise (e.g., plaintext recovery), (d) other security/integrity issue (e.g., denial of service, protocol downgrade, policy bypass).
2. Exposure duration: Time between the release of the software containing the HFB and the public disclosure of the bug.
3. Remediation effort: Difficulty of bug-fixing (i.e., patch deployment) and the revocation and/or update of compromised keys.
4. Scale: Number of affected systems before/after the release of a patch.

3 Cryptography Testing Techniques

This section provides a structured analysis of existing software testing methods and evaluates their suitability to uncover HFBs. In contrast to formally verified implementations, which can guarantee the absence of certain HFBs (as will be discussed in the next section), no single testing approach is universally effective across cryptosystems. We assess different testing methods, including differential testing, static analysis, Monte Carlo tests, fuzzing, and known answer tests, on their efficacy for the detection of HFBs. Based on this assessment, we describe a unified framework for robust cryptographic testing.

3.1 Differential Testing

Differential testing applies identical inputs to multiple independent implementations of a procedure or function to find inconsistencies. Discrepancies in the outputs indicate potential bugs caused by errors in arithmetic operations, the handling of parameters, or edge-case logic. This approach is primarily suitable to identify hard-to-find bugs in complex systems, e.g., advanced cryptographic schemes or protocols, where exhaustive specification-based testing may not be practical due to the vastness of input spaces. In the context of public-key cryptography, differential testing is applicable for deterministic operations, such as RSA decryption, RSA/ECDSA signature verification, static (EC)DH, and the validation of TLS certificates [11]. For example, the testing of RSA decryption consists of feeding identical ciphertexts and keys into different implementations and checking for consistency in the obtained plaintexts.

Limitations. Differential testing is able to detect bugs by comparing multiple implementations (without requiring a formal specification), assuming that two or more independent implementations are unlikely to share the same flaw. While it can spot basic flaws, it is less suited to detect elusive bugs that occur only in very rare edge cases, such as HFBs. Instead, differential testing is more useful for confirming bugs once discrepancies arise and to identify buggy code sections by comparing intermediate results. Furthermore, differential testing can not be (straightforwardly) applied to probabilistic operations, such as RSA encryption or signing, as it would require controlling or recording randomness.

3.2 Static Analysis

Static analysis refers to a range of techniques that analyze source code, bytecode, or binaries without execution to spot potential errors, vulnerabilities, and inefficiencies. These methods range from simple approaches, such as linting and type checking, to advanced techniques, e.g., abstract interpretation, translation of source code to mathematical models, proof-oriented equivalence checking.

In cryptographic software, static analysis plays a key role in the verification of security properties. Expressive specification languages like Cryptol can be used to describe cryptographic algorithms in a rigorous way and translate them to Satisfiability Modulo Theories (SMT) formulas, often reducing subproblems to SAT/bit-vector queries, which enables automated correctness proofs. Special tools like ct-verif focus on the verification of constant-time execution, while BinSec can detect vulnerabilities in compiled cryptographic binaries.

For example, Amazon Web Services (AWS) uses Cryptol as a specification and verification platform to translate cryptographic functions to SMT formulas for analysis by solvers. An important application of this approach is the formal verification of AWS s2n, an open-source TLS implementation that emphasizes simplicity and speed. Thanks to Cryptol and associated tools, it was possible to identify and mitigate potential vulnerabilities, such as timing side-channel attacks in the s2n HMAC implementation, prior to deployment [12].

Example: Cryptol for RSA-PSS signature generation. Given a message m or its hash $h = \mathrm{Hash}(m)$, an RSA-PSS signature is, in essence, computed as $s = (\mathrm{EMSA}(h, \mathit{salt}))^d \bmod N$, where EMSA is a probabilistic encoding method for h using a random salt. A static analysis of RSA-PSS can ensure that the encoding (including sub-operations like padding and mask generation) and the modular exponentiation are implemented correctly. With help of an appropriate tool, both a high-level specification and low-level implementation of RSA-PSS are translated into SMT formulas. The static analysis verifies the equivalence between the specification and the practical implementation for all inputs in the modeled domain. A discrepancy indicates a potential bug in, for example, the modular arithmetic, bit-manipulation (i.e., padding or encoding), or edge-case handling. The following Cryptol specification represents RSA-PSS signing:

```
rsaSign: {n} (fin n) => [n] -> [HLen] -> [SLen] -> [n] -> [n]
rsaSign d h salt N = modExp (EMSA ( h, salt )) d N
```

Here, `d`, `h`, `salt`, and `N` are a private exponent, a hash, a random value, and an RSA modulus, respectively, while `modExp` stands for a modular exponentiation and `EMSA` is the PSS encoding from RFC 8017. The Galois Software Analysis Workbench (SAW) is able to translate both the Cryptol code and a low-level implementation in, e.g., C or Java into SMT formulas. An SMT solver, such as Z3 or CVC5, then attempts to prove that this implementation adheres to the mathematical definition. The solver does this by: (a) encoding exponentiation and arithmetic constraints as bit-vector logic; (b) checking that the computed signature is always valid under all possible inputs; (c) finding counterexamples if an invalid transformation exists. The counterexamples, provided only in case the SMT solver detects a discrepancy, help to identify potential bugs related to incorrectly implemented arithmetic or incorrect padding/encoding.

Limitations. Static analysis can provide strong functional assurance (and, in some tools, constant-time execution guarantee), but its conclusions are only as reliable as the specification and the abstraction model used for verification. In particular, properties like randomness quality (entropy, bias, conditioning) and other statistical behaviors are difficult or impossible to capture in SMT-based equivalence proofs and typically require a complementary empirical validation with the help of dedicated statistical test suites, such as the tests described in the NIST special report 800-22 [27]. Finally, solver-based proofs may not scale smoothly to large, highly-optimized code bases without careful modularization and well-chosen invariants.

3.3 Monte Carlo Testing

Monte Carlo testing is a randomized testing technique that evaluates software by sampling inputs from a chosen probability distribution and checking if the executions satisfy expected properties. Unlike deterministic tests that execute a fixed (i.e., limited) set of hand-chosen cases, Monte Carlo testing can explore large input spaces (e.g., millions of cases) relatively quickly and may, thus, be able to reveal failures occurring only under particular input combinations.

When applied to cryptographic software, Monte Carlo testing usually draws random, but valid, inputs (e.g., keys, plaintexts, messages, nonces) and checks algorithm-specific invariants, such as consistency of RSA or ECDSA signature generation/verification (i.e., $\text{verify}(\text{sign}(m)) = m$), symmetry of ECDH shared secret keys (i.e., $a \cdot B = b \cdot A$, where $A = a \cdot G$ and $B = b \cdot G$), and correctness of RSA encryption/decryption round-trips (i.e., $\text{decrypt}(\text{encrypt}(x)) = x$). The distribution of input values can be uniform for wide coverage or biased toward boundary conditions (e.g., values near 0 or $p - 1$ in ECDSA and ECDH, carry-propagation boundaries, or exceptional operand encodings) in order to increase the likelihood of triggering subtle arithmetic and parsing defects.

Limitations. While Monte Carlo testing is self-contained (i.e., does not need any other implementation for reference) and may allow one to execute millions of test cases in a reasonably short time, it can still not provide a completeness guarantee. Extremely rare bugs, e.g., an HFB with a probability of 2^{-32} or even less, may remain undetected without highly targeted test-case generation.

3.4 Fuzzing

A fuzzer is an automated tool for “stress-testing” a program by executing it on a large number of randomized test cases drawn from an extended input space that covers also invalid, malformed, or out-of-specification inputs. Fuzzing can detect many kinds of defect, such as input-buffer overflows and other memory-access errors (if the fuzzer is equipped with static analysis tools, e.g., address sanitizers), incorrect input parsers, missing/insufficient input validation, and so on. These defects manifest not only via an incorrect result or output, but also through crashes, assertion violations, memory-safety errors, or other abnormal behavior. The strategy modern fuzzers adopt to produce pseudo-random inputs can be mutation-based (i.e., existing inputs are perturbed, e.g., by flipping bits or via heuristics), generation-based (i.e., inputs are constructed from grammars or models [19]), coverage-guided (i.e., inputs that explore not-yet-covered code paths are prioritized, e.g., AFL [33]), and differential (i.e., identical inputs are compared across different implementations to detect inconsistencies [31]).

In cryptographic realms, fuzzing is highly valuable for assessing robustness and error handling under incorrect or mis-formatted inputs. Public-key libraries expose many attacker-controllable parsing/decoding surfaces, e.g., ASN.1 and DER objects, X.509 certificates, PEM encodings, representations of points on an elliptic curve, signature/ciphertext formats, protocol messages. Fuzzing can probe how implementations react to NULL pointers, invalid length fields, non-canonical encodings, illegal points/scalars in ECC, malformed public keys, and unspecified parameter sets, and whether such inputs trigger unexpected results or undefined behaviour. Unlike Monte Carlo tests, fuzzing deliberately expands the input-space to inputs that violate syntactic or semantic constraints.

Example: OSS-Fuzz. A well-known example of industrial-strength fuzzing is OSS-Fuzz [15], an open-source framework developed by Google (in cooperation with the Core Infrastructure Initiative and the OpenSSF) to continuously fuzz-test open-source software. Launched in 2016, OSS-Fuzz has helped identify and fix over 10000 vulnerabilities and 36000 bugs across 1000 open-source projects (as of August 2023). Among those were hundreds of cryptography-related bugs found in major libraries like OpenSSL, LibreSSL, GnuTLS, libgcrypt, NSS, and Crypto++. For example, in 2017, OSS-Fuzz discovered multiple vulnerabilities of critical severity in GnuTLS, including CVE-2017-5334, a memory corruption caused by a double-free bug in the X.509 certificate parsing routine. This bug is exploitable from remote via a specifically crafted X.509 certificate that contains a Proxy Certificate Information extension and could, in the worst case, allow an attacker to crash an application using GnuTLS.

Limitations. Many functional HFBs are semantic (e.g., subtle arithmetic defects) and do not manifest as a crash, hang, memory leak, or other unexpected behaviour; without strong oracles (reference models, invariants, or differential checks), fuzzing may miss such flaws even if it executes the relevant code. Another limitation is that detecting common side-channel vulnerabilities, such as key-dependent execution times or memory access patterns, requires specialized measurement and analysis beyond normal fuzzing, and often calls for dedicated tools. Finally, most modern security protocols are stateful (e.g., handshakes in TLS, especially in the case of a renegotiation), whereby “interesting” states can only be reached with sequences of well-formed messages, which requires special fuzzers that are protocol/state-aware rather than purely input-centric [13].

3.5 Known Answer Tests (KATs)

KATs are an essential tool for validating the correctness of a software function (or a full program or system) by checking whether it reproduces a set of known input-output pairs (“test vectors”) generated with a reference implementation or specification. Such test-vectors, along with meta-data, are normally stored in hex-format in special KAT files. Unlike differential testing, KATs neither need an “on-the-fly” generation of (pseudo-)random numbers for input data nor the execution of other implementations for output-checking. KAT-based testing is very easy to automate and provides strong regression protection: once a bug is fixed, the corresponding vector is kept to prevent a re-introduction. KATs also support portability (they can find architecture/compiler-dependent deviations in an effective way) and continuous integration (fast pass/fail oracles).

KATs for cryptographic software can serve different purposes, one of which is to determine the correctness of an implementation and its compliance with standards. For example, the NIST provides KAT collections for this purpose as part of their Cryptographic Algorithm Validation Program (CAVP). A second purpose of KATs is to detect HFBs via special test vectors targeting defects in parsing, arithmetic, and boundary handling that may hide in rare edge cases.

Example: Wycheproof. Wycheproof [16], originally developed by Google, is an open-source cryptographic test suite providing KATs to trigger subtle, real-world implementation pitfalls, with emphasis on rare edge cases and historical bug patterns. Whenever a new critical bug or vulnerability in a cryptographic library is reported, corresponding test cases are added to Wycheproof to catch this specific bug and prevent it from appearing in other libraries. Wycheproof's test functions were initially written in Java and later ported to Go, though the test-vectors themselves are language-agnostic [16]. We provide a C-port of the public-key test functions in Wycheproof along with this paper to facilitate the testing of C-based libraries. This port is available on GitHub [30] and enhances the accessibility of Wycheproof without altering its core methodology.

Limitations. Despite their value, KATs are inherently incomplete as they can not cover the full input space: passing a finite set of vectors does not imply the correctness for all inputs. KATs can miss defects that require a specific internal state, long execution history, or extremely rare operand patterns. Even KATs specifically tailored to HFBs, such as Wycheproof, can only find already known bugs, i.e., bugs that have been publicly disclosed.

3.6 Framework for Cryptographic Testing

Given the rarity and subtlety of HFBs, an effective testing environment has to deliberately target the very specific execution conditions where such bugs often hide. We propose a 3-layered approach combining deterministic, statistical, and dynamic techniques, each optimized for HFB discovery.

– *Layer 1: HFB-tailored KATs.* KATs represent the most precise layer of the framework and check an implementation using curated input-output pairs aimed at triggering historical or structurally plausible HFBs. Unlike generic KATs, HFB-specific test-vector suites like Wycheproof deliberately include malformed or non-canonical inputs, and boundary values to cause overflows or carry-mispropagations in multi-precision arithmetic.
– *Layer 2: Monte-Carlo testing with biased distributions.* While KATs are an effective tool to detect known HFBs, they fail to uncover previously unseen flaws. Monte-Carlo testing can address this gap using (valid) inputs drawn from deliberately biased distributions that concentrate probability mass on carry-propagation boundaries, near-0/near-p values in finite-field and group operations, edge-case encodings and borderline valid points or scalars, and atypical but still standards-compliant domain parameters. Such systematic testing of functions around arithmetic or structural "danger zones," where HFBs naturally arise, increases the chance of triggering unknown defects.
– *Layer 3: Continuous coverage-based fuzzing.* This layer ensures robustness and covers not only the core cryptographic operations but also "peripheral functions," which are often complex and can contain many execution paths (e.g., X.509 parsing). Fuzz-testing is especially effective at catching HFBs related to input validation, encoding/decoding, and error-handling.

4 Verified Cryptographic Implementations

Even the most advanced testing regime can not guarantee that a cryptographic software implementation is bug-free. Formal verification can come to the rescue and further improve correctness and security by providing strong (in the sense of mathematically grounded) assurances. In this section, we delve into formal verification methods and the utilization of modern 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 a given implementation adheres to formally specified correctness and security features across all possible executions. Unlike conventional testing, which only inspects a finite subset of possible inputs, formal verification provides global guarantees by exhaustively analyzing every possible execution path. This process ensures the absence of flaws, such as arithmetic errors, incorrect memory handling, and logic inconsistencies, that could compromise security [8].

The verification procedure begins with the formal specification of a cryptographic function $f : \mathcal{X} \to \mathcal{Y}$ and its desired properties. Taking RSA encryption as a simple example, correctness is typically expressed as

$$\forall (pk, sk), \forall m \in \mathcal{M}, \quad \mathrm{Dec}(sk, \mathrm{Enc}(pk, m)) = m. \tag{1}$$

This property, a global invariant, has to hold under all conditions. Verification tools like Coq, EasyCrypt, and Tamarin Prover translate both the formal specification and an actual software implementation into logical representations (based on first or higher-order logic) and attempt to establish their equivalence through automated theorem proving or model checking [2,4,6].

In addition to cryptographic algorithms, formal verification techniques have also been extensively applied to various security protocols, ensuring properties such as confidentiality, integrity, and authenticity. One common approach uses the symbolic Dolev-Yao model, which abstracts the cryptographic primitives as perfect black boxes while modeling adversarial interactions. Tools such as ProVerif and Tamarin Prover automate the verification by analyzing protocol logic to detect vulnerabilities. For example, ProVerif has been instrumental in verifying the security of some TLS 1.3 draft variants, identifying weaknesses and validating subsequent security enhancements [5].

Challenges and Limitations. Despite its strengths, formal verification faces some challenges. The high complexity of cryptographic schemes and protocols may lead to state-space explosion, making an exhaustive analysis computationally expensive. In addition, ensuring that a formal model accurately represents a real-world implementation is critical, as discrepancies can enable undetected vulnerabilities. Furthermore, formal methods struggle to capture side-channel attacks and compiler-introduced optimizations (e.g., vectorization), which calls for complementary empirical validation techniques, such as differential analysis and fuzz-testing [8]. In summary, while formal verification is a powerful tool to prove the correctness of an implementation, it is most effective in combination with empirical testing to achieve comprehensive security validation.

4.2 Memory Safety and Type Safety

Memory safety is paramount in software since vulnerabilities can lead to severe security breaches. A recent information sheet published by the NSA mentions that a significant portion of software vulnerabilities, namely around 70%, stem from memory safety issues in languages like C, C++, and Assembly [26]. The NSA recommends adoption of memory-safe programming languages to mitigate these risks, particularly in security-critical domains.

Rust is considered as 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 operations, where any unintended memory access can impair security. Formally, the Rust borrow checker ensures that, for any memory state $\mathcal{M}$, a Rust function f will execute safely:

$$\forall \mathcal{M}, \quad \text{BorrowCheck}(f, \mathcal{M}) = \text{safe}. \tag{2}$$

This guarantee is crucial for cryptographic implementations, where unintended memory access can lead to security breaches. Furthermore, Rust's type system enforces that operations are performed on compatible types, which reduces the likelihood of logical errors within cryptographic algorithms. Low-level libraries like `subtle` facilitate the development of constant-time code, essential for preventing timing attacks by ensuring that its operations (e.g., conditional moves and selections, comparisons) have no data-dependent timing variations. Several cryptographic crates, such as `ring` [29], `rustls`, and `RustCrypto`, leverage the memory-safety and type-safety features of Rust. Alternatives like Go sacrifice performance for garbage collection, making them less suitable for applications with low-latency or real-time requirements.

Rust typically exhibits a slight performance penalty compared to C code in cryptographic primitives, but nonetheless remains competitive with aggressive optimization. Benchmarks, such as those for implementations of AES, indicate that Rust approaches C's performance, with any overhead largely coming from higher-level abstractions rather than intrinsic limitations [28]. In contrast, C's minimalism usually provides an advantage in resource-constrained settings like embedded systems [20]. While C excels at low-level efficiency and binary code-size, Rust's richer abstractions, simplified memory management, and extensive libraries can inflate size and complexity if unoptimized, though its optimization potential occasionally surpasses that of C by leveraging advanced integration across function and library boundaries. In addition, 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 to translate Rust into formal languages, e.g., Coq and F*, for security-critical applications) [23,17]. These capabilities are absent in C since the C ecosystem lacks direct verification support. Thus, selecting between Rust's safety and C's raw performance requires a careful trade-off analysis.

4.3 Low-Level Optimization and Verification

Jasmin [1] aims to bridge (high-level) cryptographic specifications and verified Assembly implementations through a certified compiler (`jasminc`) and formal verification in EasyCrypt. Its core syntax mirrors low-level control flow while enabling mathematical reasoning about correctness and side-channel resistance of an implementation. The methodology to verify code is as follows.

1. Modeling: Translate the Assembly code to Jasmin (this preserves low-level optimizations).
2. Annotation: Add pre-conditions, post-conditions, and loop invariants in the form of EasyCrypt predicates.
3. Equivalence checking: Use the CryptoLine toolchain to prove equivalence between the Jasmin code and the reference models via algebraic predicates (e.g., $\texttt{mont\_asm}(x) \equiv \texttt{mont\_ref}(x) \bmod p$).
4. Compiler certification: Rely on the correctness of the `jasminc` compiler to ensure that the generated Assembly code matches Jasmin semantics.

This approach combines automated SMT solving (for range checks) and interactive theorem proving (for modular equivalences), which allows for verification of both arithmetic correctness and side-channel security. Using single-precision modular reduction for lattice-based cryptography as example, we demonstrate how to employ Jasmin for two critical tasks: ensuring constant-time arithmetic and proving equivalence between optimized code and reference models.

Verified Modular Reduction. Consider Montgomery's reduction technique for a prime modulus p, where $R = 2^{64} > p$ and $p' = -p^{-1} \bmod R$. The Jasmin implementation computes $\mathrm{Mont}(x) = (x + (x \cdot p' \bmod R) \cdot p)/R$, which always has to satisfy $\mathrm{Mont}(x) \equiv x \cdot R^{-1} \bmod p$ and $0 \leq \mathrm{Mont}(x) < 2p$. Below is the Jasmin code 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
}
```

The `assert` clauses specify a pre- and post-condition. EasyCrypt proves the equivalence between this code and the mathematical specification via symbolic execution and modular arithmetic lemmas. The `jasminc` compiler guarantees "constant-timeness" by rejecting branching on secrets.

Jasmin remains very relevant today, playing a crucial role in post-quantum cryptography (especially lattice-based schemes) and growing its impact beyond integer arithmetic. For example, it turned out to be useful in uncovering issues in an implementation of the Falcon signature algorithm with emulated floating-point arithmetic. Namely, Jasmin helped verify a problem in the multiplication function where intermediate products are zeroized prematurely, which violates IEEE 754 rounding rules [18]. The verification process revealed that, while 692 out of 2048 FFT constants could trigger incorrect zeroization, the impact was mitigated in practice due to lower bounds on intermediate values.

In summary, the combination of formal verification techniques with modern programming languages designed for safety and performance, such as Rust and Jasmin, offers a robust foundation for the development of secure cryptographic implementations. Rust's guarantees of memory and type safety, and Jasmin's support for low-level optimization and formal verification, enable developers to write cryptographic software that is efficient and provably secure. Using these methods—optionally complemented by the testing framework described in the previous section—represents the current "best practice" for the implementation of cryptographic software.

5 HFBs Collection

As mentioned at the end of Sect. 1, this paper is supplemented by some online resources, in particular a collection of HFBs that have been discovered in real-world cryptographic software, including established libraries like OpenSSL. The HFB collection is publicly available on GitHub [30]. At the time of writing this paper, the GitHub repository contained data on more than 50 HFBs, which we collected from numerous open-source projects by studying bug reports, revision histories, discussion forums, mailing lists, research articles, results from large-scale testing/fuzzing initiatives (e.g., Wycheproof, CryptoFuzz), databases like CVE, and various other online resources. Most of the bugs slipped through the testing regime applied by the corresponding projects, and many actually ended up in production. However, not all of these HFBs led to serious vulnerabilities because their impact is often mitigated by system redundancies or the absence of single-point-of-failure dependencies.

This section provides a brief overview of the information about each HFB contained in the repository. The HFBs are summarized in six main Markdown files, roughly corresponding to six main categories of bugs as follows.

- `CARRY_PROPAGATION.md`: Mishandling of carry or borrow bits (resp., chains of carries or borrows) in multi-precision integer arithmetic.
- `CRYPTO_STATE.md`: Incorrect updates to a (secret-dependent) program state or context of a cryptosystem or protocol.

- `IMPLEMENTATIONS.md`: Deviations from formal mathematical or algorithmic specification (other than carry-propagation flaws).
- `INPUT_VALIDATION.md`: Incorrect (incomplete) validation of inputs that are invalid, malformed, or otherwise manipulated.
- `PARAM_HANDLING.md`: Incorrect validation or handling of domain parameters (including encoding/decoding errors).
- `CONSTANT_TIME.md`: Secret-dependent branches or memory access patterns that thwart constant-time execution.

Each bug contained in the repository is documented via the following basic format: *(1) Specification:* description of the bug's context and affected software component(s); *(2) Defect:* the exact defect caused by the bug; *(3) Impact:* the consequences of the defect, especially in terms of cryptographic correctness and potential security risks; *(4) Code Snippet:* an excerpt of the source code where the bug manifests. This structured documentation helps to better understand the nature of HFBs, trace their origins, and identify patterns that can improve future testing and verification efforts. Some examples of HFBs are described in the full version of this paper, which is publicly available on GitHub in the same repository [30] as the collection of HFBs.

5.1 Categorization of HFBs

We group the HFBs into six major categories that recurred across many of the affected cryptographic software projects. Each class captures a distinct failure mode, i.e., a different way the underlying cryptographic guarantees fail. There exist, however, some overlap cases where a bug fits in two categories, e.g., when a bug causes an erroneous computation and also enables a timing attack.

Carry-Propagation Flaws. These bugs arise when carry-bits or borrow-bits are not propagated properly across the words/limbs of multi-precision integers and hide in functions for arithmetic in a prime field or multiplicative group, such as addition, subtraction, multiplication, and squaring, usually including reduction modulo a prime p. Moreover, auxiliary operations like the conversion between full and reduced-radix representation and the final conversion of field-elements into canonical form can be affected. Because triggering inputs often sit at word (resp., limb) boundaries or have highly-specific values (e.g., near 0 or p), these bugs can easily escape conventional software tests.

State or Context Mismanagement. Here, the implementation maintains a state or a context that governs the basic properties and behavior of a cryptographic algorithm (e.g., context of nonce-generation for ECDSA, configuration flag in ECDH indicating whether or not a key can be re-used) or protocol (e.g., state of a TLS session renegotiation), but the state is not initialized and/or updated properly. The resulting failure can be sporadic, manifesting only under specific input sequences or renegotiation patterns, and present in a wrong output, the execution of an exception handler, or abnormal program termination.

Incorrect Implementation. These bugs cause an implementation to deviate from the mathematical or algorithmic specification. A common root-cause are flaws in arithmetic operations, such as exponentiation, scalar multiplication, addition and doubling of elliptic-curve points, and their "low-level" modular operations (other than carry-propagation issues). They can yield wrong results for a small subset of inputs, e.g., acceptance of a carefully-crafted adversarial signature.

Missing or Insufficient Input Validation. Input parsers and API front-ends fail to thoroughly check sizes, lengths, encodings, required parameters, or specific bounds (e.g., ASN.1/DER fields, maximum message sizes). Other examples are non-canonical (or otherwise invalid) exponents or scalars, and group elements (points) of low order. The consequences can range from an overread/overwrite of input buffers to acceptance of ill-formed inputs that should be rejected.

Insecure Domain Parameters. An implementation erroneously accepts insecure or invalid domain parameters (e.g., subgroup generators of wrong order, "prime fields" whose cardinality is actually not a prime) or interprets the parameters inconsistently across different configurations or code paths. Such inconsistencies are always problematic in heterogeneous environments where different libraries interoperate, and can enable attacks through subtle parameter misuse.

Timing-Attack Vulnerabilities. They are caused by secret-dependent branches (e.g., if-then clauses in exponentiation, scalar multiplication, or modular arithmetic) or secret-dependent memory-access patterns (e.g., table lookups).

5.2 Data Collection and Statistics

Collecting, categorizing, and analyzing HFBs discovered in real-world cryptographic software provides insights on subtle implementation pitfalls and traps that even experienced developers may fall into. Apart from the documentation of each HFB, we also generated some statistics, which we hope turn out to be useful when tackling the question of why and how HFBs arise.

For each bug, a statistical record containing the following information was assembled (see file `HFB.csv` in the GitHub repository [30]): *(1) ID*: The unique identifier for the bug, which can be a CVE number or a specific project-related identifier like a Git commit-ID; *(2) Category:* A high-level categorization of the bug, such as "Carry", "State", or "Timing", based on the six HFB categories outlined in the previous subsection; *(3) Language:* The programming language of the affected implementation, e.g., "Assembly" or "C"; *(4) SubType:* A more granular classification of the bug, such as "Montgomery Squaring" for an issue in a function for Montgomery squaring; *(5) Impact:* A description of the bug's immediate effect(s), such as "Incorrect result", "Memory corruption", or "Side-channel"; *(6) Severity:* A numerical value indicating the severity of the bug on a scale from 1 to 10, based on, e.g., CVE severity scores.

A (preliminary) analysis of the statistics we collected about the 50+ HFBs contained in our GitHub repository yields the following observations:

1. Complexity of Assembly: A disproportionately large share of bugs occurred in hand-written or tool-generated Assembly code, whereby readability and maintainability were often sacrificed for performance.
2. Prevalence of carry-propagation flaws: Bugs caused by the mis-propagation of carry/borrow bits are among the most common, highlighting that multi-precision arithmetic operations are error-prone, in particular when they are aggressively optimized for speed.
3. Cross-platform inconsistencies: Slight differences in compiler behavior, the supported instruction set(s) of the CPU (e.g., BMI2, ADX), or endianness caused bugs that remained undetected in single-target test environments.

6 Conclusions

In this survey paper, we delivered a thorough examination of HFBs in cryptographic software, with a specific emphasis on public-key cryptosystems. These subtle bugs pose a number of unique challenges to the security and robustness of cryptographic software due to their low probability of occurrence, high complexity, and resistance to detection by standard testing methods. Our analysis underlines the urgent need to tackle these elusive bugs and further investigate the vulnerabilities and attacks they enable.

We systematically evaluated a spectrum of software testing methodologies tailored to uncover HFBs, including differential testing, static analysis, Monte Carlo testing, fuzzing, formal verification, and KATs. Each methodology offers distinct advantages: differential testing excels at spotting inconsistencies across different libraries; static analysis provides rigorous defect detection and ensures properties like "constant-time" execution; Monte Carlo testing offers statistical input coverage; fuzzing reveals robustness issues with malformed inputs; formal verification delivers strong mathematical assurances of correctness; and KATs (exemplified by tools like Wycheproof) serve as an essential baseline validation by testing against known HFBs and various edge cases. Our accompanying C-port of Wycheproof makes its extensive KAT-collection available for platforms that are not supported by Go, most notably embedded devices.

To advance HFB detection, we proposed a test framework that integrates KATs, Monte Carlo testing, and continuous fuzzing. This approach combines deterministic, statistical, and dynamic testing benefits, empowering developers to bolster the security of their cryptographic software effectively. However, this may still not be sufficient for extremely 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, carefully categorized to illuminate their highly diverse nature and aid in mitigation efforts. We hope this collection will serve as a vital resource for the cryptographic community, offering real-world examples that highlight common pitfalls and reinforce the necessity of specialized testing. By documenting these bugs—ranging from carry propagation flaws to constant-time failures—we aim to improve testing strategies and prevent recurring vulnerabilities.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, et al. Jasmin: High-assurance and high-speed cryptography. In *24th ACM Conference on Computer and Communications Security (CCS 2017)*, pages 1807–1823. ACM, 2017.
2. J. B. Almeida, M. Barbosa, G. Barthe, et al. The last mile: High-assurance and high-speed cryptographic implementations. In *41st IEEE Symposium on Security and Privacy (S&P 2020)*, pages 965–982. IEEE, 2020.
3. D. F. Aranha, F. R. Novaes, A. Takahashi, et al. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *27th ACM Conference on Computer and Communications Security (CCS 2020)*, pages 225–242. ACM, 2020.
4. G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM Symposium on Principles of Programming Languages (POPL 2009)*, pages 90–101. ACM, 2009.
5. K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *38th IEEE Symposium on Security and Privacy (S&P 2017)*, pages 483–502. IEEE, 2017.
6. B. Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust — POST 2012*, pages 3–29. Springer, 2012.
7. J. Blessing, M. A. Specter, and D. J. Weitzner. Cryptography in the wild: An empirical analysis of vulnerabilities in cryptographic libraries. In *19th ACM Asia Conference on Computer and Communications Security (ASIACCS 2024)*, pages 605–620. ACM, 2024.
8. B. Boston, S. Breese, J. Dodds, et al. Verified cryptographic code for everybody. In *Computer Aided Verification — CAV 2021*, pages 645–668. Springer, 2021.
9. P. Bressana, N. Zilberman, and R. Soulé. Finding hard-to-find data plane bugs with a PTA. In *16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2020)*, pages 218–231. ACM, 2020.
10. B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *Topics in Cryptology — CT-RSA 2012*, pages 171–186. Springer, 2012.
11. Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, pages 793–804. ACM, 2015.
12. A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook. Continuous formal verification of Amazon s2n. In *Computer Aided Verification — CAV 2018*, pages 430–446. Springer, 2018.
13. J. De Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USS 2015)*, pages 193–206. USENIX Association, 2015.
14. Fail0verflow. Console hacking 2010: PS3 epic fail. Presentation at the 27th Chaos Communication Congress (27C3), 2010.
15. Google. OSS-Fuzz: Continuous fuzzing for open source software. `https://github.com/google/oss-fuzz`, 2020.
16. Google. Project Wycheproof. `https://github.com/google/wycheproof`, 2020.
17. Hax Team. Hax: A Rust verification toolchain for security-critical software. `https://github.com/hax-rust/hax`, 2023.
18. V. Hwang. Formal verification of emulated floating-point arithmetic in Falcon. In *Advances in Information and Computer Security — IWSEC 2024*. Springer, 2024.

19. S. Jero, M. L. Pacheco, D. Goldwasser, and C. Nita-Rotaru. Leveraging textual specifications for grammar-based fuzzing of network protocols. In *31st Conference on Innovative Applications of Artificial Intelligence (IAAI 2019)*, pages 9478–9483. AAAI Press, 2019.
20. D. Kasak. Rust vs. C: A performance comparison in systems programming. Blog post, `https://deniskasak.github.io/rust-vs-c-perf`, 2018.
21. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96*, pages 104–113. Springer, 1996.
22. D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail? A case study and open problems. In *5th Asia-Pacific Workshop on Systems (APSys 2014)*, pages 7:1–7:7. ACM, 2014.
23. Libcrux Team. Libcrux: A formally verified cryptographic library for Rust. `https://github.com/cryspen/libcrux`, 2023.
24. S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.
25. N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker. Finding bugs in cryptographic hash function implementations. *IEEE Transactions on Reliability*, 67(3):870–884, Sept. 2018.
26. National Security Agency. Software memory safety. Cybersecurity information sheet, NSA, 2022. `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`.
27. A. Rukhin, J. Soto, J. Nechvatal, et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication 800-22, National Institute of Standards and Technology (NIST), 2010.
28. T. Seaborn. Performance analysis of RustCrypto: AES implementations in Rust vs. C. `https://rustcrypto.org/performance`, 2019.
29. B. Smith. Ring: Safe, fast, small crypto using Rust. `https://briansmith.org/rustdoc/ring/`, 2023.
30. M. Steinbach. Wycheproof-C: A C cryptographic test suite. `https://github.com/mattc-try/wycheproof-c/`, 2025.
31. G. Vranken. Differential fuzzing of cryptographic libraries. `https://archive.is/https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/`, 2019.
32. R.-P. Weinmann. Assessing and exploiting bignum vulnerabilities. BlackHat 2015, `https://comsecuris.com/slides/slides-bignum-bhus2015.pdf`, 2015.
33. M. Zalewski. Technical whitepaper for AFL-fuzz. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2014.
34. M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, Mar. 2009.