Eidolon: Perform Noise-Aware Fuzzing on FHE Libraries via Equivalence Expression Transformation

ZHENSHENG XIAN, Tsinghua University, China
ZHEN YAN, Tsinghua University, China
YUANLIANG CHEN, Tsinghua University, China
XUELIAN CAO, Tsinghua University, China
FUCHEN MA*, Tsinghua University, China
DALONG SHI, AVIC International Digital Network Technology Co., Ltd., China
YU JIANG*, Tsinghua University, China

Ensuring data privacy during computation is a critical challenge in many security systems. Fully Homomorphic Encryption (FHE) addresses this gap by enabling multiple operations on encrypted data without decryption, thus ensuring privacy is preserved throughout computation. However, existing cryptographic testing tools are unable to test the core functionality of FHE, which is the execution of computations on encrypted data. They are expertly designed to generate structured data for testing cryptographic algorithms. This structural mismatch, combined with a lack of awareness of FHE-specific noise management, leads them to generate invalid test inputs that fail to probe FHE libraries' core logic.

To address this gap, we propose Eidolon, a noise-aware fuzzer. It directs mutations toward arithmetic expressions that explore the computational space defined by the noise budget. As its test oracle, Eidolon leverages Equivalence Expression Transformation, which transforms a standard arithmetic expression into two mathematically identical but structurally different forms (e.g., Factored, Horner) to detect inconsistencies in their outputs. We evaluated Eidolon on SEAL, OpenFHE, HElib, and TFHE. Compared with existing cryptographic and grammar-based fuzzers, Eidolon achieves 28.7%, 45.5%, 75.6%, and 37.6% higher final code coverage than CLFuzz, Cryptofuzz, CDF, and Peach, respectively. In total, Eidolon uncovered 20 previously unknown bugs, 10 of which have been fixed and 12 assigned CVEs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Noise-Aware Testing

ACM Reference Format:
Zhensheng Xian, Zhen Yan, Yuanliang Chen, Xuelian Cao, Fuchen Ma, Dalong Shi, and Yu Jiang. 2026. Eidolon: Perform Noise-Aware Fuzzing on FHE Libraries via Equivalence Expression Transformation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE102 (July 2026), 22 pages. https://doi.org/10.1145/3808109

1 Introduction

While widely used cryptographic libraries such as OpenSSL [38] and SymCrypt [33] effectively protect data at rest and in transit, their limitations in safeguarding data during computation pose

*Fuchen Ma and Yu Jiang are the corresponding authors.

Authors' Contact Information: Zhensheng Xian, KLISS, BNRist, School of Software, Tsinghua University, China; Zhen Yan, KLISS, BNRist, School of Software, Tsinghua University, China; Yuanliang Chen, KLISS, BNRist, School of Software, Tsinghua University, China; Xuelian Cao, KLISS, BNRist, School of Software, Tsinghua University, China; Fuchen Ma, KLISS, BNRist, School of Software, Tsinghua University, China; Dalong Shi, AVIC International Digital Network Technology Co., Ltd., China; Yu Jiang, KLISS, BNRist, School of Software, Tsinghua University, China.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE102
https://doi.org/10.1145/3808109

an important security challenge, particularly in untrusted cloud environments. FHE addresses this challenge by enabling computations on encrypted data [50]. This capability ensures that sensitive information remains protected throughout computation, making FHE a promising technology for applications ranging from secure financial analysis [1] to privacy-preserving visual search [2].

In FHE, noise refers to the random perturbation intentionally introduced into ciphertexts during encryption to ensure security. This noise gradually accumulates during homomorphic computations, and once it exceeds a threshold, decryption fails. Managing this noise significantly increases the complexity of FHE implementations, making it difficult to avoid bugs. For example, a bug in OpenFHE [45] caused additions in the CKKS scheme to return incorrect results. Such bugs violate the correctness guarantees of FHE, silently corrupt results, and undermine the security of applications that rely on them. Since FHE libraries are critical to data privacy, bugs in their implementations can have serious consequences, ranging from incorrect computation results to potential leakage of sensitive information.

Ensuring the correctness of FHE libraries is therefore an important security concern, raising the key question of whether existing cryptographic testing tools are adequate for this new cryptographic paradigm. In recent years, fuzzing has emerged as a promising method for detecting bugs; it can automatically generate diverse inputs to test software. Cryptographic fuzzing tools have been widely explored for testing cryptographic algorithm implementations, such as Cryptofuzz [49] and CDF [4], using differential fuzzing to compare outputs across different library implementations. Building on this foundation, CLFuzz [52] extends it into a semantic-aware fuzzer, thereby enabling more efficient and comprehensive vulnerability detection. Additionally, grammar-based fuzzers such as Peach [35] can generate syntactically valid inputs but remain unaware of FHE-specific noise constraints.

However, while these fuzzing methods are effective for improving the correctness of cryptographic algorithms and generating syntactically valid inputs, they are limited when applied to FHE libraries, as they lack the ability to produce well-formed inputs necessary for comprehensively testing FHE computations. Also, the lack of a reliable oracle for FHE further limits the detection of subtle implementation bugs.

To effectively detect bugs in FHE implementations, there are two main challenges: (1) **The first challenge** is to generate valid and high-quality arithmetic expressions as inputs. At the same time, it is necessary to generate a value (e.g., an x) to instantiate these expressions, such as $x^2 + 1$, so that they can be executed in FHE computations. The feasible computations form a computational space bounded by the available noise budget. As long as the noise remains below the threshold, arbitrary arithmetic expressions can in principle be evaluated. Without effective awareness, most generated cases will either be invalid or too shallow to explore the full computational space. (2) **The second challenge** is to design a reliable test oracle for FHE implementations. Differential testing is limited in effectiveness for FHE implementations, as vast differences in FHE parameters, noise management, and implementation-specific optimizations across libraries make it difficult to perform differential testing across FHE libraries. This necessitates testing within a single-implementation context, where the core difficulty lies in distinguishing a genuine implementation bug from an acceptable result of the library's internal noise management. Without a clear ground truth, determining whether an unexpected result reflects a vulnerability or an acceptable outcome of noise growth is challenging.

To address these challenges, we propose Eidolon, a noise-aware fuzzing framework designed to systematically uncover bugs in FHE libraries. First, Eidolon leverages the remaining noise budget after each computation as feedback to guide the mutation of subsequent arithmetic expressions. By doing so, it effectively navigates the computational space and reaches deeper expressions that would otherwise remain unexplored. Second, to overcome the oracle challenge without relying

on an external ground truth, Eidolon introduces an Equivalence Expression Transformation technique. This technique transforms a base arithmetic expression into mathematically equivalent but structurally distinct variants, such as Standard, Factored, and Horner forms, that impose different workloads on FHE libraries. By comparing the decrypted outputs of these variants against each other and against a native baseline executed in the programming language, Eidolon provides a robust oracle for detecting inconsistencies.

We implemented Eidolon[1] and evaluated its effectiveness on four widely used FHE libraries: OpenFHE [5], SEAL [42], HElib [30], and TFHE [46]. Compared with existing cryptographic and grammar-based fuzzers, the 24-hour results show that Eidolon detects 13, 15, 20, and 14 more bugs, and achieves higher code coverage of 28.7%, 45.5%, 75.6%, and 37.6% than CLFuzz [52], Cryptofuzz [49], CDF [4], and Peach [35], respectively. In total, Eidolon has found 20 previously unknown bugs (10 in OpenFHE, 3 in SEAL, 3 in HElib, and 4 in TFHE), of which 12 have been assigned CVE identifiers due to their severity. In summary, we make three key contributions:

- We propose a noise-aware fuzzer that leverages the noise budget as feedback to systematically explore computational spaces in FHE.
- We employ the Equivalence Expression Transformation technique to establish a reliable oracle that detects bugs by comparing the outputs of mathematically equivalent but structurally distinct arithmetic expressions.
- We implement and evaluate Eidolon on four widely used FHE implementations. It detects 20 previously unknown bugs. Among them, 10 have been fixed, and 12 have been assigned CVEs.

2 BACKGROUND

2.1 Fully Homomorphic Encryption Concept

Fully Homomorphic Encryption (FHE) [19, 50] is a cryptographic paradigm that enables computation on encrypted data without decryption. As illustrated in Figure 1, the general FHE computation process involves translating a high-level arithmetic expression into a sequence of FHE API calls that operate on encrypted data. Specifically, each operation consumes the ciphertext's noise budget (e.g., reducing it from 56 to 14 after multiplication), and the final result can be successfully decrypted only if the residual noise (e.g., 8) remains within the valid budget.

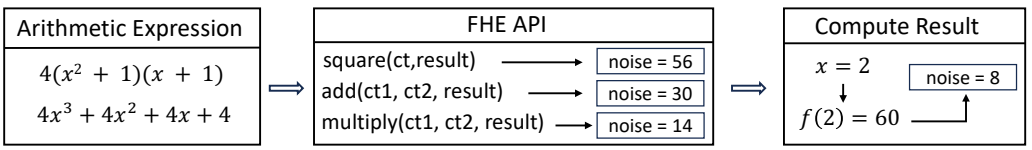

Fig. 1. The general computation process of FHE. High-level arithmetic expressions are translated into API calls (e.g., multiply, add) that consume the noise budget.

To support these diverse demands, several schemes have been developed, each optimized for different data types and operations. Modern FHE libraries typically implement one or more of the following schemes: TFHE [14] is optimized for Boolean circuits and gate-by-gate computations; BFV and BGV [7, 8, 17] target exact integer arithmetic over finite fields or integer rings, suitable for applications such as private database queries; CKKS [11, 13] supports approximate arithmetic on real and complex numbers, making it efficient for privacy-preserving machine learning.

[1] https://anonymous.4open.science/r/Jsd2wwewq0sdaSvsFds35nxsjai/

Beyond these schemes, the FHE landscape includes other important designs and optimizations. For example, the **GSW (Gentry-Sahai-Waters)** [22] family offers slower noise growth, ideal for deep circuits, while techniques like the **Residue Number System (RNS)** [6, 12] are used to accelerate schemes like CKKS. This continuous evolution of diverse and specialized FHE approaches underscores the need for a general and adaptable testing methodology.

The security and functionality of these schemes are governed by a complex set of parameters that must be carefully chosen to balance three competing requirements: security, correctness, and performance. Key parameters, often denoted as a set $\mathcal{P} = \{N, q, t, ...\}$, typically include the **polynomial modulus degree (N)**, which is a primary determinant of the security level. Another crucial pair of parameters are the **ciphertext modulus (q)** and the **plaintext modulus (t)**. The plaintext modulus t defines the space where the original data is encoded (e.g., integers modulo t), while the much larger ciphertext modulus q defines the space for encrypted data and is directly related to the initial computational capacity, known as the **noise budget**. Selecting a valid and efficient parameter set is a non-trivial task; an improperly chosen set can lead to either insufficient security or a noise budget so small that even simple computations fail.

2.2 Noise Mechanism in FHE

A characteristic of FHE schemes is the presence of cryptographic 'noise' [16, 22] in every ciphertext. This noise is essential for security, but it also creates a limitation on computation. At the heart of this mechanism is the concept of a **noise budget**.

In FHE, noise refers to the small, random error intentionally added to a ciphertext during encryption. Each freshly encrypted ciphertext is endowed with a finite computational capacity, its noise budget, which is determined by the scheme's parameters. Every homomorphic operation—particularly multiplication—consumes a portion of this budget. In so-called **leveled schemes** [21], such as BFV or CKKS, this budget is designed to support a pre-determined computational depth. If the cumulative operations exhaust the budget, the noise overflows, irreparably corrupting the ciphertext and leading to decryption failure or incorrect results. Consequently, the entire challenge of performing complex FHE computations revolves around managing this finite resource before it is depleted. For simplicity, we will hereafter use the term *noise* to refer to the remaining noise budget of a ciphertext.

To overcome this limitation and enable computations of arbitrary depth, FHE schemes employ a powerful technique known as **Bootstrapping** [20, 27]. Bootstrapping is a process that effectively resets a ciphertext's noise, restoring its budget to a fresh state. While this technique allows for a virtually unlimited number of operations, it is a computationally expensive procedure. Its application varies across different schemes. In schemes like TFHE, it is a frequent and integral part of the computation, whereas in the aforementioned leveled schemes, it is used more sparingly as a powerful but costly tool to extend computations beyond their pre-set depth limit.

The correctness of an FHE computation is thus tied to the intricate mechanics of noise management. A fuzzer oblivious to noise will generate overwhelmingly invalid inputs, preventing it from exploring any meaningful computational space. Therefore, leveraging the noise as feedback is a powerful strategy for systematically exploring the deep and complex computation space in FHE.

3 Motivation

Certain bugs in FHE libraries are difficult to detect because they can lead to incorrect results without causing overt failures like system crashes. This challenge is further complicated because differential testing is hard to apply to FHE, and behavioral inconsistencies exist even between schemes with similar functionality (e.g., BFV and BGV) within the same library.

```
1 EncryptionParameters parms(scheme_type::BFV); // Parameter Setup
2 ...
3 vector<int64_t> vect(slot_count, -3);
4 Plaintext plain;
5 batch_encoder.encode(vect, plain); // Encode Operation
6 Ciphertext cipher;
7 encryptor.encrypt(plain, cipher); // Encrypt Operation
8 evaluator.multiply_plain_inplace(cipher, plain); // Output [-258029, -233196, 291981,...]
9 // it suppose be [9, 9, 9, ...]  ⟶ Bug
```

Fig. 2. Example of an unexpected result in SEAL BFV multiplication. Multiplying a ciphertext by a plaintext encoded with negative values produces an incorrect output compared to the expected result.

Bug Example in SEAL BFV. Figure 2 illustrates a bug in the *multiply_plain_inplace*() API of the Microsoft SEAL library [43]. This issue occurs when handling plaintext vectors containing negative values. In lines 3-4, a vector of negative integers (e.g., −3) is encoded into a plaintext using the BatchEncoder. At line 12, the evaluator.*multiply_plain_inplace*() function is invoked to perform ciphertext–plaintext multiplication. The expected decrypted result should be [9, 9, 9, ...], but instead it produces incorrect values such as [-258029, -233196, 291981, ...]. This indicates an implementation flaw in handling negative plaintexts. Moreover, the same issue can also be triggered when using the *multiply_plain* API, showing that the bug consistently affects ciphertext-plaintext multiplication with negative plaintext.

Why Existing Tools Fail to Detect This Bug. Existing cryptographic fuzzing tools struggle to uncover such bugs, since the FHE libraries allow for multiple computations in ciphertext, whereas existing cryptographic fuzzers are designed to generate structured inputs that satisfy the requirements of cryptographic algorithms for differential testing. Figure 3 shows an example input generated by Cryptofuzz and CLFuzz under the SEAL BFV scheme. To trigger this bug, it is necessary to generate an input x with a negative value, construct the expression $x * plain(x)$, and transform it into a concrete API call. Neither CLFuzz nor Cryptofuzz is capable of generating this arithmetic expression. Even if CLFuzz and Cryptofuzz cover the code lines containing the bug, they fail to activate the required execution scenario. Figure 4 presents the function signatures of two relevant APIs. Cryptofuzz, which uses a coverage-guided approach to generate and mutate inputs, may reach these functions and even invoke them, but it still cannot construct the computation path needed to expose the bug. CLFuzz attempts to extract semantic information from function signatures. Still, the generated inputs cannot be combined into arithmetic expressions that trigger the bug, and thus cannot reach the computation path. Therefore, even with semantic awareness, these tools remain unable to reproduce execution scenarios that expose this type of bug.

```
1 Running: SEAL BFV
2 Input x: (-9857, -23678, 0)
3 Arithmetic Expression: None
```

(a) Cryptofuzz or CLFuzz

```
1 Running: SEAL BFV
2 Input x: (23819, 3920, -574)
3 Arithmetic Expression: (2x+1, x^2+5, x^3+1)
```

(b) Eidolon

Fig. 3. Illustrative examples of inputs generated for SEAL BFV: (a) Cryptofuzz or CLFuzz, (b) Eidolon.

How to Detect Such Bugs. Detecting this bug requires shifting the focus from testing cryptographic algorithms to testing FHE computations. To trigger bugs such as the one illustrated in Figure 2, a fuzzer must generate not only valid ciphertext inputs but also arithmetic expressions (e.g., Figure 3 (b)) and translate them into concrete API calls. This is essential for exercising the core computational paths of FHE libraries. In addition, a reliable oracle is needed to distinguish acceptable results from genuine bugs. For example, the expression $x^2 + 2x + 1$ is mathematically equivalent to $(x + 1)^2$, yet their execution in an FHE library may follow different paths, consuming noise differently and stressing the implementation in distinct ways. By comparing the outputs of such equivalent forms after decryption, as well as against a native result computed directly in the programming language, inconsistencies can be detected without relying on an external cross-library ground truth, thereby revealing hidden bugs.

```
1 void multiply_plain(const Ciphertext &encrypted, const Plaintext &plain, Ciphertext
      &destination, MemoryPoolHandle pool = MemoryManager::GetPool()) const
2
3 void multiply_plain_inplace( Ciphertext &encrypted, const Plaintext &plain, MemoryPoolHandle
      pool = MemoryManager::GetPool()) const;
4 }
```

Fig. 4. Function signatures of *multiply_plain* and *multiply_plain_inplace* in SEAL. These methods take a ciphertext and a plaintext as input, and produce the multiplication result in the ciphertext, with an optional memory pool parameter.

Lesson Learned. From this case, we can draw two important lessons: (1) To effectively increase the probability of discovering such bugs, an FHE fuzzer must efficiently explore the computational space by constructing valid arithmetic expressions while avoiding noise overflow. This requires not only generating inputs that pass basic parameter validation, but also systematically managing the depth and structure of arithmetic expressions so that they remain executable within the constrained noise. Without such noise-aware guidance, the fuzzer would overwhelmingly generate invalid test cases due to noise overflow, wasting computational resources and failing to exercise the deeper functionalities of FHE libraries where subtle bugs are more likely to emerge. (2) The ambiguity between incorrect computation and noise-induced decryption failure creates a barrier to validating FHE implementations. Differential testing across libraries (e.g., SEAL vs. OpenFHE) is often impractical due to divergent APIs and design choices, making it difficult to create comparable testing environments. This necessitates a reliable oracle for bug detection that operates within a single library.

4 DESIGN

Design Goal: A practical noise-aware fuzzer tailored specifically for FHE libraries should have the following properties:

- General: Eidolon is designed to find bugs across a wide range of FHE schemes, ranging from integer-based schemes (e.g., BFV, BGV) and approximate arithmetic schemes (e.g., CKKS) to Boolean-circuit-based schemes (e.g., TFHE). It requires only minimal adaptation across different library implementations.
- Efficient: Eidolon maximizes the generation of valid test cases, enabling it to efficiently explore the vast computational space of FHE libraries and uncover 20 previously unknown bugs within 24 hours.

- Reliable: Eidolon integrates an oracle to accurately distinguish genuine implementation flaws from expected failures due to noise overflow. This enables the automatic generation of a concise Minimal Working Example for each vulnerability found, providing developers with definitive evidence of the bug.

4.1 Eidolon Workflow

Figure 5 illustrates the workflow of Eidolon, driven by two main phases that operate in an iterative loop: Test Case Construction and Noise-Aware Fuzzing. This workflow is designed to systematically expose bugs in FHE implementations.

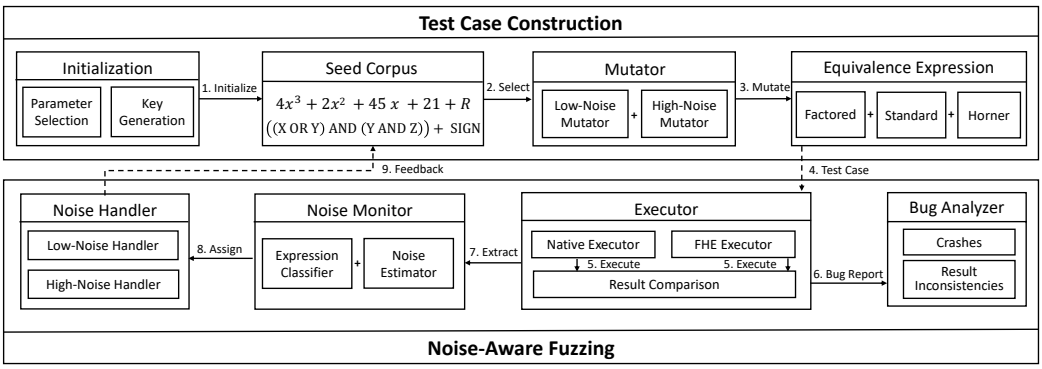

Fig. 5. The workflow of Eidolon. It consists of two main phases: (1) Test-Case Construction, which generates arithmetic expressions and corresponding inputs, and (2) Noise-Aware Fuzzing, which iteratively explores the computational space under noise constraints. A feedback loop connects both phases, ensuring that informative seeds are preserved and reused to drive deeper exploration and bug discovery.

Test Case Construction: (1) Eidolon first initializes a consistent FHE context by selecting key security parameters, including plaintext modulus, etc. The corresponding secret, public, and evaluation keys are then generated. (2) It selects a standard arithmetic expression from an evolving seed corpus (e.g., $x^2 + 1$ or $3x - 2$) as the basis for constructing the new arithmetic expression. (3) Based on noise from previous computations, Eidolon chooses either a High-Noise mutator or a Low-Noise mutator and applies it to the standard expression, producing a new arithmetic expression. (4) This expression then undergoes the Equivalence Expression Transformation, generating its two mathematically equivalent forms (Horner and Factored). These two forms, along with the Standard expression, now constitute a set of test cases ready for execution.

Noise-Aware Fuzzing: (5) The Native Executor evaluates the arithmetic expression over plaintext to obtain a baseline result. In parallel, the FHE Executor performs the homomorphic computation, producing a ciphertext result. (6) The decrypted FHE result is compared against the native result. Any discrepancies, crashes, or anomalies detected during this consistency check are reported as potential bugs. (7) The Noise Monitor then assesses the remaining noise of the resulting ciphertext. (8) This noise information is dispatched to a corresponding handler. If the remaining noise is low (indicating a high-consumption, difficult-reach test case), the Low-Noise Handler is invoked. It rewards the seed by prioritizing it for future mutations and signals for fine-tuned exploration. Conversely, if the noise is high (indicating a low-consumption, easy-reach test case), the High-Noise Handler is triggered to guide the mutation toward more complex expressions. This feedback loop, managed by the handlers, directly determines which mutator is selected in the next

iteration. (9) Finally, the updated seed, along with its associated feedback information, is returned to the Seed Corpus. This closes the feedback loop by ensuring that seeds leading to valuable execution scenarios are preserved and reused in subsequent fuzzing iterations. The process is repeated until a predefined time budget or iteration limit is reached.

4.2 Test Case Construction

The selection of FHE parameters plays a critical role in governing the noise, thereby defining the computational capacity of the scheme. To systematically detect implementation bugs, Eidolon begins its process by configuring the FHE context using the parameter set recommended by the library. An initial seed expression is then selected from a corpus. This seed is subsequently processed by the noise-aware mutator. To create an oracle for detecting bugs, the mutated expression is then transformed into computationally equivalent forms, such as Horner and Factored forms. By evaluating and comparing these structurally diverse but mathematically equivalent expressions, Eidolon comprehensively explores the library's behavior at its operational limits, enabling the reliable detection of implementation bugs.

Initialization: At the beginning of the fuzzing process, Eidolon initializes a *FHE context*, C, which is defined by a set of FHE parameters $\mathcal{P} = \{N, q, t, ...\}$. These parameters govern the security and computational properties of the FHE scheme, directly influencing noise growth. Based on $\mathcal{P}$, a key tuple $\mathcal{K} = \{sk, pk, ek\}$ is generated. The context $C = (\mathcal{P}, \mathcal{K})$ remains fixed throughout the fuzzing process to ensure that result variations arise from the computations being tested rather than the underlying setup. It is important to note that while these fundamental parameters are common, their specific naming, configuration APIs, and available ranges may differ significantly across FHE libraries.

Seed Corpus Management: The seed corpus, $\mathcal{S}$, is a dynamically evolving set of arithmetic expressions, $\mathcal{S} = \{s_1, s_2, ..., s_n\}$, used as the starting point for mutation. It is initialized from two primary sources to ensure representativeness and diversity. First, valid arithmetic sequences are extracted from the official unit tests and usage examples of the target FHE libraries, reflecting developer-intended usage patterns. For example, from SEAL's example code, we extracted operations such as ciphertext-plaintext multiplication (e.g., `multiply_plain(x, 4)`), rotation operations (e.g., `rotate_rows(x, 3)`), and composite expressions (e.g., `square(x + 1)`), which exercise fundamental computation paths and demonstrate typical usage patterns. Second, we supplement these with a collection of manually constructed expressions selected for their structural diversity. These seeds encompass a broad spectrum of operator combinations and expression topologies. A key challenge is to ensure the corpus is continuously enriched with computationally diverse seeds, rather than being populated with trivial or redundant expressions. To address this, Eidolon employs a feedback-driven prioritization strategy. The guiding principle behind this strategy is that arithmetic expressions that successfully execute while inducing either very high or very low noise consumption are valuable for exploration, as they explore the boundaries and sparse regions of the computational space. Therefore, during the fuzzing process, if a mutated expression s' executes without causing a noise overflow, it is considered for inclusion. Priority is then given to those expressions deemed interesting by the noise handlers, ensuring that $\mathcal{S}$ is continuously enriched with high-value seeds.

Noise-Aware Mutation: The central challenge in mutation is to efficiently explore the vast computational space without being dominated by invalid test cases caused by noise overflow. Eidolon addresses this by adapting the mutation strategy based on noise feedback. Let η_0 be the initial noise of a fresh ciphertext, and let $\eta(s_{FHE})$ be the residual noise after an FHE evaluation of a mutated seed s'. We define two noise thresholds: a lower bound $\tau_{low} = 0.1 \cdot \eta_0$ that triggers fine-grained exploration, and an upper bound $\tau_{high} = 0.8 \cdot \eta_0$ that marks the transition into high-noise

regions of the computational space. The mutation strategy M, for a seed expression $s \in \mathcal{S}$ is chosen as follows:

$$M(s) = \begin{cases} \text{Low-Noise Mutator}, & \text{if } \eta(s'_{FHE}) < \tau_{low} \\ \text{High-Noise Mutator}, & \text{if } \tau_{low} \leq \eta(s'_{FHE}) < \tau_{high} \text{ or } \eta(s'_{FHE}) \geq \tau_{high} \end{cases}$$

The High-Noise Mutator applies aggressive transformations to rapidly increase computational complexity. These operations include inserting new operations that consume more noise (e.g., multiplications such as $x \rightarrow x \cdot y$) and magnifying coefficients (e.g., $3x \rightarrow 10x$), both of which accelerate noise consumption to reach deeper logic. Conversely, the Low-Noise Mutator performs fine-grained modifications to carefully explore the neighborhood of a seed. These operations include inserting operations with lower noise impact (e.g., additions such as $x \rightarrow x + y$), removing operations to reduce depth (e.g., $x^2 + x + 1 \rightarrow x + 1$), and reducing coefficients (e.g., $4x \rightarrow 2x$). This mutation strategy allows Eidolon to balance exploration and exploitation of the noise-constrained computational space.

Equivalence Expression Transformation: Once a new arithmetic expression is generated through noise-aware mutation, Eidolon requires a robust oracle to verify the correctness of its homomorphic evaluation. Simply comparing a single FHE-computed result against the native result is insufficient; such a check cannot reliably distinguish a genuine implementation bug from an expected decryption failure caused by noise overflow. To solve this, Eidolon introduces the Equivalence Expression Transformation technique, which creates a self-contained oracle by transforming a single expression into multiple, structurally different but mathematically identical forms. This method is inspired by program transformation techniques [44, 51] and leverages algebraic properties.

We selected Factored and Horner forms because they represent distinct differences in multiplicative depth and operation order, which are the primary factors influencing FHE noise consumption. While simpler transformations such as commutativity ($a + b \rightarrow b + a$), associativity ($a + (b + c) \rightarrow (a + b) + c$), and nested grouping ($x^2 + 2x + 1 \rightarrow (x^2 + 2x) + 1$) exist, they cause negligible changes to the noise compared to these structural alterations. For a given standard arithmetic expression, Eidolon generates two primary variants. The first is the Factored Form, where the expression is converted into its factored equivalent (e.g., $x^2 + 2x + 1$ becomes $(x + 1)^2$). This form often reduces the number of multiplications, altering the computational depth and noise consumption profile. The second variant is derived using Horner's Method [15], transforming it into a nested structure (e.g., $ax^3 + bx^2 + cx + d$ becomes $d + x(c + x(b + xa))$). This transformation minimizes the total number of multiplications to the degree of the expression, creating a sequential dependency that stresses the FHE library's handling of intermediate results.

By executing the Standard, Factored, and Horner forms homomorphically, Eidolon creates three distinct computational paths for the same underlying mathematical function. The core principle is that while the intermediate noise consumption paths differ, the final decrypted results must be mathematically identical. These three FHE results are then compared against both each other and the native computed result. This differential comparison allows for the detection of subtle inconsistencies, turning the challenge of bug detection into a robust consistency check. A bug is flagged only when a clear deviation pattern emerges (e.g., Standard and Factored both decrypt to 35 while the native result is 36), effectively filtering out noise failures. This ensemble of expressions, along with the native evaluation plan, forms the final test case sent to the execution phase.

4.3 Noise-Aware Fuzzing

Without noise awareness, random mutations often produce arithmetic expressions that exceed the ciphertext's computational capacity. Eidolon addresses this limitation by using noise as a

feedback signal to guide mutations, ensuring they explore diverse noise patterns while maintaining computational validity. Figure 6 illustrates how mutations without noise awareness quickly exceed the noise, whereas noise-guided mutations systematically explore different noise regions.

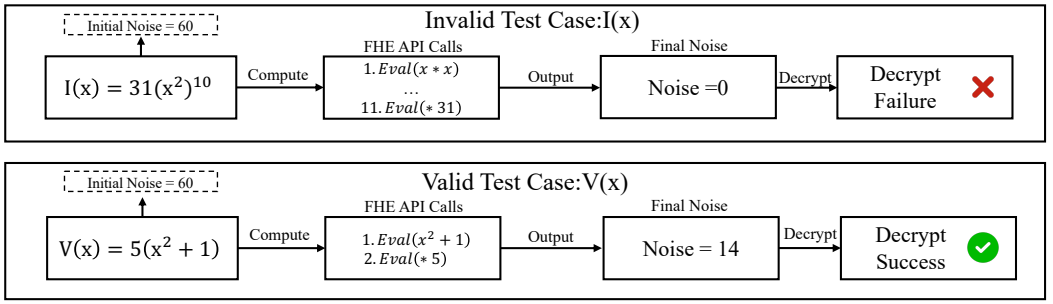

Fig. 6. Impact of noise-awareness on FHE fuzzing. Starting from the same initial noise, the invalid case (top) requires a long sequence of FHE API calls with high multiplicative depth, exhausting the noise and causing decryption failure. The valid case (bottom) balances operations with fewer API calls, preserving enough noise for a successful decryption.

The example contrasts two mutation outcomes starting from the same initial noise (Initial Noise = 60). The invalid case (top), $I(x) = 31(x^2)^{10}$, is translated into a long sequence of FHE API calls dominated by multiplications (e.g., $\texttt{Eval}(x * x)$, ..., $\texttt{Eval}(*31)$). As the figure shows, this high multiplicative depth completely exhausts the noise (resulting in Noise = 0), which inevitably causes a Decrypt Failure. Without noise-awareness, traditional mutation methods often produce a high ratio of such invalid test cases, leading to inefficient testing and additional computational overhead. In contrast, the valid case (bottom), $V(x) = 5(x^2 + 1)$, requires only two FHE API calls ($\texttt{Eval}(x^2 + 1)$, $\texttt{Eval}(*5)$), consuming noise more slowly and preserving enough noise for a Decrypt Success (leaving Noise = 14). Eidolon is designed to learn from these outcomes, penalizing mutations that lead to noise overflow like in $I(x)$ and prioritizing balanced structures like $V(x)$ to efficiently explore the valid computational space.

Noise Monitor: The Noise Monitor serves as the primary sensor of the fuzzing loop. After the FHE Executor completes a computation, the monitor assesses the remaining noise of the resulting ciphertext. It then quantifies this value relative to the initial noise (η_0) and categorizes the noise level based on two predefined thresholds, τ_{low} and τ_{high}. This categorized information is then dispatched to the appropriate handler.

Noise Handler: The Noise Handler is the strategic core of the fuzzing process, translating raw noise into directives that govern mutation strategy and seed prioritization. The mechanism is implemented through two complementary handlers that work in tandem to guide Eidolon's focus. The High-Noise Handler operates by default, directing Eidolon to continue using the High-Noise Mutator to search for complex computational paths. If a test case is found to be too simple, yielding a residual noise above the τ_{high} threshold, this handler lowers the priority of its seed. When a test case successfully pushes the computation to its limits, causing the noise to drop below τ_{low} , the Low-Noise Handler is triggered. It immediately designates the test case as high-priority and directs a switch to the Low-Noise Mutator for focused, fine-grained exploration of this critical boundary.

Noise-Aware Exploration: Algorithm 1 presents the noise-aware fuzzing process. After initializing the FHE context and selecting a base expression from the seed corpus (Lines 1-5), Eidolon iteratively mutates the base expression using either the Low-Noise or High-Noise mutator based on the residual noise η_{last} from the previous execution (Lines 6-10), and applies the Equivalence

Algorithm 1: Noise-Aware Fuzzing

Input: C : FHE Context
$expr_{base}$: base arithmetic expression
Output: B: set of discovered bugs

1 $B = \{\}$;
2 $C.Initalize(Parameters)$;
3 $KeyGenerate(C)$;
4 $\eta_{last} \leftarrow Initial\ ciphertext\ noise$;
5 $expr_{base}$ = SelectFromCorpus($\mathcal{S}$);
6 **while** *true* **do**
7 **if** $\eta_{last} < \tau_{low}$ **then**
8 $expr_{Standard} = LowNoiseMutator(expr_{base})$;
9 **else**
10 $expr_{Standard} = HighNoiseMutator(expr_{base})$;
11 **end**
12 $\{expr_{Horner},\ expr_{Factored}\} = Transform(expr_{Standard})$;
13 $result_{native} = Native\_Executor(expr_{Standard})$;
14 $result_{FHE} = FHE\_Executor(expr_{i\in\{\text{Std, Hor, Fac}\}}, C)$;
15 $\eta_{last} = \eta_{FHE}$;
16 $newBugs = checkOracle(res_{native}, res_{FHE})$;
17 **if** $newBugs$! = $NULL$ **then**
18 $B.append(newBugs)$;
19 **else if** *Decryptable* **then**
20 $\mathcal{S}.append(expr_{standard})$;
21 **end**

Expression Transformation to produce the Horner and Factored variants (Line 12). All three forms are evaluated by the FHE Executor and compared against the native baseline; discrepancies are reported as potential bugs, and successful expressions are appended to the seed corpus to close the feedback loop (Lines 13-20).

4.4 Bug Detection and Analysis

For runtime error detection, Eidolon leverages compiler-level instrumentation (AddressSanitizer [25], UndefinedBehaviorSanitizer [26]) together with the FHE libraries' internal assertions. Any sanitizer report, failed assertion, or unhandled exception is classified as a crash bug, and the triggering input and expression are saved.

For correctness bugs, Eidolon uses an oracle based on Equivalence Expression Transformation. Formally, we define a correctness bug as a violation where the decrypted homomorphic evaluation deviates from the native plaintext execution:

$$\text{Dec}(\text{Eval}(E, \text{Enc}(x))) \neq E(x)$$

where E is the arithmetic expression and x is the input. It executes three mathematically equivalent forms (Standard, Horner, Factored) homomorphically and compares them against a native baseline. Computations exceeding the noise can exhibit stochastic decryption failures. To distinguish these decryption failures from deterministic implementation bugs, Eidolon employs a re-execution

strategy to deduplicate bugs from false positives. Specifically, a single mismatch is initially tolerated; the system flags only those discrepancies that persist upon re-execution or appear consistently across multiple forms. This strategy reduces false positives from 1,284 to 148, as only 1.6% of initial mismatches persisted upon re-execution in our evaluation. Confirmed bugs are reported with the triggering arithmetic expression, API call sequence, and FHE parameters, which serve as the basis for constructing a Minimal Working Example.

5 IMPLEMENTATION

We implemented Eidolon in the four widely-used open-source FHE libraries: OpenFHE (version 1.2.1 [37]), Microsoft SEAL (version 4.1.2 [43]), HElib (version 2.3.0 [29]) and TFHE (version 1.0.1 [46]). Our implementation includes both the core fuzzing logic and dedicated adaptors to interface with each library's unique API.

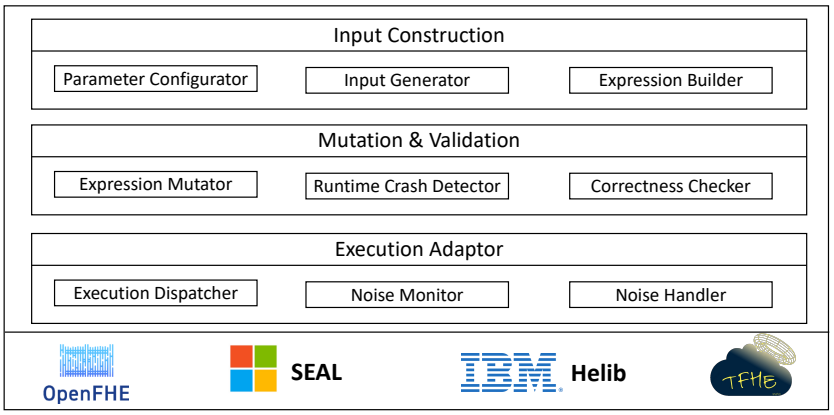

Fig. 7. Components of Eidolon, including Input Construction, Mutation & Validation, and Execution Adaptor

Figure 7 presents the components of Eidolon, which can be divided into three core parts. The first part is Input Construction, responsible for initializing the FHE context, managing the seed corpus, and constructing arithmetic expressions as test inputs. The second part is Mutation and Validation, which mutates expressions under noise feedback and validates results using the oracle based on equivalent transformations. The third part is Execution Adaptor, which provides a uniform interface to different FHE libraries, executes the constructed tests, monitors noise, and collects coverage and bug reports.

Fuzzing Engine. We implemented Eidolon in roughly 3,000 lines of C++ code on top of LibFuzzer [32], an in-process, coverage-guided fuzzing framework.

Schemes Diversity. These FHE libraries differ in the schemes they support. OpenFHE supports BFV, BGV, CKKS, and also integrates the TFHE scheme. Microsoft SEAL provides efficient implementations of BFV, BGV, and CKKS, often with hardware acceleration. HElib focuses on BGV and CKKS, while TFHE is dedicated to the TFHE scheme. Among these, BGV [8], BFV [17], TFHE [14], and CKKS [13] represent the most practical and widely adopted FHE schemes today.

Noise Monitor. In FHE, the process of querying the residual noise requires library-specific adaptors due to API diversity. For libraries like Microsoft SEAL and HElib, we utilize their provided APIs, such as `invariant_noise_budget()` and `capacity()`, to directly measure the noise. OpenFHE is more fine-grained; we employ its static noise estimation mechanism for the BFV and BGV schemes, while for the CKKS scheme, we use its dynamic noise estimation. For any library or scheme lacking

Table 1. Summary of bugs discovered in four widely used FHE libraries. Eidolon identified 20 bugs in total, and 12 of them have been assigned CVE identifiers. We reported all the bugs to the corresponding developers, and 10 of them have been fixed and 13 are confirmed. We hide the CVE identifiers for double-blind reviews.

#	Libraries	Bug Description	ID	Bug Status
1	OpenFHE	Integer overflow when $(2^{scaleModSize} * i)$ equals 2^{63} due to incorrect comparison operators.	CVE-2024-50669	Fixed
2	OpenFHE	Signed integer overflow when computing modular exponentiation.	CVE-2025-28193	Fixed
3	OpenFHE	Segmentation fault in BINFHE Decrypt function due to missing null pointer validation.	CVE-2025-28182	Fixed
4	OpenFHE	SIGSEGV in BINFHE EvalSign function due to missing null pointer validation.	CVE-2025-28183	Fixed
5	OpenFHE	SIGSEGV in BINFHE BTKeyGen function due to missing null pointer validation.	CVE-2025-28186	Fixed
6	OpenFHE	SIGSEGV in BINFHE EvalNOT function due to missing null pointer validation.	CVE-2025-28190	Fixed
7	OpenFHE	SIGSEGV in BINFHE EvalDecomp function due to missing null pointer validation.	CVE-2025-28189	Fixed
8	OpenFHE	SIGSEGV in BINFHE EvalBinGate function due to missing null pointer validation.	CVE-2025-28184	Fixed
9	OpenFHE	SIGSEGV in BINFHE EvalFloor function due to missing null pointer validation.	CVE-2025-28187	Fixed
10	OpenFHE	SIGSEGV in BINFHE EvalFunc function due to missing null pointer validation.	CVE-2025-28185	Fixed
11	HElib	Memory leak in BGV mode during Context initialization with bootstrappable configuration.	CVE-2025-28196	Confirmed
12	HElib	Heap-buffer-overflow in BGV mode in context initialization with bootstrappable configuration.	CVE-2025-28195	Confirmed
13	HElib	Segmentation fault in HElib logging due to null pointer reference binding in ostream.	BUG#521	Reported
14	SEAL	Invalid reference initialization in IterTuple construction due to missing appropriate constructor.	BUG#719	Confirmed
15	SEAL	Undefined behavior in CKKS scheme due to infinity value conversion to int type.	BUG#732-1	Fixed
16	SEAL	Undefined behavior in CKKS scheme due to infinity value conversion to unsigned long type.	BUG#732-2	Fixed
17	TFHE	Signed integer overflow in lweSymEncryptWithExternalNoise when computing Torus32.	BUG#271-1	Reported
18	TFHE	Signed integer overflow in lweCreateKeySwitchKey, which exceeds int32_t range.	BUG#271-2	Reported
19	TFHE	Signed integer overflow in tGswAddMuIntH, which exceeds int32_t range.	BUG#271-3	Reported
20	TFHE	A heap buffer overflow occurs in lweCopy function.	BUG#271-4	Reported

a direct noise-querying interface, Eidolon defaults to a correctness-guided inference strategy, using the success or failure of a 'test decryption' to determine the ciphertext's validity.

6 EVALUATION

To evaluate the effectiveness of Eidolon, we compared it with four testing tools: the cryptographic fuzzers CLFuzz, Cryptofuzz, and CDF, and the grammar-based fuzzer Peach. These experiments were conducted on four widely used FHE libraries. To ensure a fair and statistically robust comparison, we established a consistent experimental setup. We configured each library with its standard parameter sets. Crucially, we enforced semantic consistency across the initial seed corpora of all tools. Specifically, the arithmetic expressions from Eidolon's initial seeds were mapped into the corresponding API call sequences or grammar formats required by each baseline tool. All experiments were repeated 30 times on the same server. We employed the Vargha-Delaney A12 measure [47] to quantify effect size, where 0.50 indicates no difference and ≥ 0.71 indicates a large effect size, and the two-tailed Mann-Whitney U test [3] to determine statistical significance at the 0.05 level. We designed experiments to address the following research questions:

- RQ1: Can Eidolon effectively find previously unknown bugs in widely used FHE libraries?
- RQ2: Does Eidolon boost the generation of valid test inputs?
- RQ3: Can Eidolon achieve higher code coverage compared to existing fuzzing tools?

6.1 Bugs in FHE Libraries

Eidolon discovered 20 previously unknown bugs across the four FHE libraries: 3 in SEAL, 10 in OpenFHE, 3 in HElib, and 4 in TFHE. Of these, 13 have been confirmed, with 10 already fixed. Detailed information on these bugs is presented in Table 1.

Bug Exploitation. The bugs discovered by Eidolon have practical security implications. Specifically, Bug#3-#11 and Bug#13 can be exploited to conduct DoS attacks by causing process crashes or

resource exhaustion. Bug#1, #2, #15, #16, and #17 involve integer overflows and undefined behaviors that can cause silent incorrect computations. Furthermore, memory safety vulnerabilities in Bug#12 and Bug#20 create the potential for arbitrary code execution.

Comparison with Existing Fuzzers. We further evaluated the performance of baseline fuzzers in detecting these 20 specific bugs. As summarized in Table 2, Eidolon successfully detected all 20 bugs. In contrast, CLFuzz detected only 7 bugs, Peach found 6, Cryptofuzz found 5, and CDF failed to detect any. This significant gap highlights that existing cryptographic fuzzers and grammar-based fuzzers, which lack noise-awareness and equivalence oracles, struggle to reach the deep computational states required to trigger FHE-specific bugs. Across 30 repeated experiments, Eidolon consistently detected all 20 bugs in every trial, achieving a Mann-Whitney U test *p-value* of < 0.001 and an effect size A_{12} of 1.0 against all baselines. We also evaluated the detection capabilities of sanitizers. ASAN and UBSAN detected 8 of the 20 newly discovered bugs (#11, 12, 15, 16, 17, 18, 19, 20). However, they failed to detect the remaining 12 bugs, which are FHE-specific logic errors such as incorrect noise management, wrong computation, and parameter handling. Eidolon successfully uncovered these bugs by leveraging its noise-aware exploration and equivalence expression oracle.

Table 2. Bugs found by Eidolon and other SOTA fuzzers. Existing SOTA fuzzers detect no more than 7 bugs, while Eidolon detects 20 unknown bugs.

Tool	Bug Number	Bugs ID#
Eidolon	20	#1 - #20
CLFuzz	7	#3, 11, 14, 17, 18, 19, 20
Cryptofuzz	5	#14, 17, 18, 19, 20
CDF	0	–
Peach	6	#11, 14, 17, 18, 19, 20

Bug Study 1: Incorrect Result on Boundary Inputs in OpenFHE CKKS Evaluation. Eidolon discovered a bug in the OpenFHE library (CVE-2024-5xxx9) that led to silent computation errors when handling boundary inputs. This bug was triggered by our oracle, which detected an inconsistency between the FHE-computed result and the native result when evaluating a particular arithmetic expression $31x^2 + 32x + 1$.

Root Cause: The issue arises from a flawed boundary check within the CKKS scheme's internal multiplication logic. During a scaling operation, an intermediate value could reach the maximum value of the signed integer type, causing it to wrap around into a large negative number. The subsequent conditional logic contained an off-by-one error: values exceeding the maximum were handled correctly, but the exact-limit case was not. This bug caused the boundary case input to follow an incorrect code path and produce erroneous results without raising any errors.

Bug Study 2: Mishandling of Boundary Values in SEAL CKKS Encoding. Eidolon discovered two bugs in the Microsoft SEAL library (Bugs #15 and #16), both occurring within the CKKS encoding process when handling extreme floating-point values. These bugs, which lead to program crashes, highlight flaws in the library's input validation and data conversion routines and were triggered by generating input vectors containing values at the edge of the standard double data type's representable range.

Root Cause: The bugs stem from the encoder's failure to safely handle exceptional numerical inputs before performing sensitive operations. The first arises in the rounding phase during a floating-point to unsigned integer conversion, where an extremely large input produces a value outside the representable range and triggers undefined behavior. The second appears in an earlier

log2 calculation during parameter checking, where a denormalized floating-point value close to zero causes an overflow.

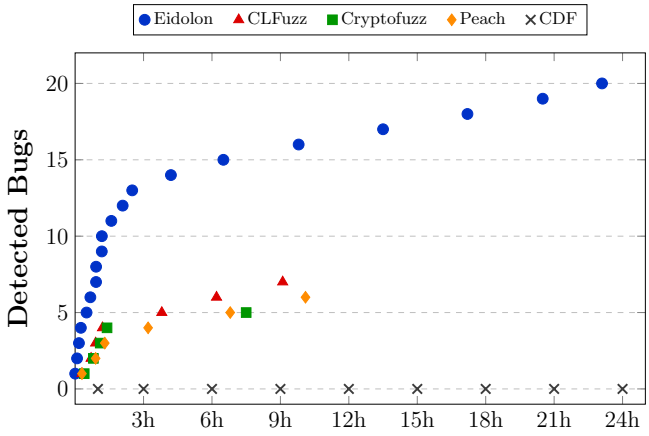

Fig. 8. Number of bugs detected by Eidolon and other SOTA fuzzers over time.

Bug Discovery Timeline. As shown in Figure 8, Eidolon discovers bugs faster than all baselines, uncovering the majority within the initial hours while other tools plateau early. Eidolon detected all 20 previously unknown bugs within the 24-hour experiments, whereas CLFuzz detected 7 bugs, Peach detected 6 bugs, Cryptofuzz detected 5 bugs, and CDF detected none. CLFuzz and Peach showed no further discovery after approximately 9 and 7 hours, respectively. This sustained discovery confirms the effectiveness of Eidolon's noise-aware strategy in navigating the FHE computational space.

Table 3. Historical bug reproduction rates across different fuzzers. Results are shown as (Detected/Total).

Tool	OpenFHE	SEAL	HElib	TFHE	Total
Eidolon	2/3	2/3	1/2	2/2	7/10
CLFuzz	0/3	1/3	1/2	2/2	4/10
Cryptofuzz	0/3	1/3	1/2	2/2	4/10
CDF	0/3	0/3	0/2	0/2	0/10
Peach	0/3	1/3	1/2	2/2	4/10

Analysis of False Negatives. To assess potential false negatives, we evaluated the ability of all tools to rediscover 10 known historical bugs from previous library versions. Across 30 repeated 24-hour runs for each case, the results were consistent. As shown in Table 3, Eidolon successfully reproduced 7 out of 10 bugs, outperforming the best baseline, which only detected 4. Among the 7 bugs reproduced by Eidolon, ASAN and UBSAN detected 5, while the remaining 2 are FHE-specific correctness bugs. For the pairwise comparison between Eidolon and the best-performing baselines, the Mann-Whitney U test yields a $p\text{-}value < 0.001$ and the A12 measure is 1.0, indicating that Eidolon's outperformance is statistically significant. The 3 missed bugs highlight the limitations of Eidolon. Two were flaws in non-computational APIs (e.g., key-switching) outside our oracle's scope, and the third was a serialization bug. These findings are further discussed in Section 7.

Answer to RQ1: Eidolon is effective at discovering bugs in FHE library implementations. In total, 20 previously unknown bugs were discovered in four widely used FHE libraries, with 12 of them assigned CVE identifiers in the US National Vulnerability Database.

6.2 Effectiveness of Noise-Aware Exploration

To assess the effectiveness of Eidolon in improving valid input generation, we conducted a comparative study between three mutation strategies within Eidolon: Noise-Aware, Coverage-Guided, and Random. This experiment measures how effective each strategy is at producing valid inputs. We excluded the TFHE library from this and the subsequent coverage analysis (Section 6.3) due to its smaller codebase and, more importantly, a critical bug we found that caused persistent crashes, preventing continuous fuzzing.

- **SIC** (Supported Input Count): The number of test inputs that can be successfully encrypted into valid FHE ciphertexts by the target library.
- **VIC** (Valid Input Count): The total number of test inputs that can complete the homomorphic computation process and be successfully decrypted to obtain a result.

Valid Input Generation We record the VIC values for each strategy once SIC reaches 10,000. As shown in Table 4, for each FHE library, the proportion of valid inputs generated by Eidolon is higher than Coverage-Guided and Random strategies. Compared with Coverage-Guided and Random mutations, Eidolon achieves improvements of 1.34× and 3.57× on average across all libraries. We set the SIC threshold to 10,000 to account for the different execution speeds across libraries.

Table 4. Comparison of valid input generation across different mutation strategies. Results are measured among 10,000 generated inputs.

Library	SIC	VIC-Random	VIC-Coverage	VIC-NoiseAware	Improvement
OpenFHE	10,000	3274	8874	9435	2.87× / 1.06×
SEAL	10,000	2387	6122	9214	3.86× / 1.51×
HElib	10,000	2456	6658	9766	3.98× / 1.47×

By using noise as feedback to guide the mutation of arithmetic expressions, Eidolon systematically generates a higher proportion of valid inputs, thereby increasing the probability of triggering more bugs and enabling broader exploration of the computation space, which improves the overall effectiveness of the fuzzing process.

Baseline Tools Valid Input Comparison. To compare the valid input generation capability across different tools, we measured the total test cases generated and valid input ratio for all fuzzers during the 24-hour experiments. For baseline tools, a valid input is defined as a test case that successfully completes execution without causing a noise-induced decryption failure. As shown in Table 5, Eidolon consistently achieves the highest valid input ratio **(89.2%–92.1%)** across all libraries, outperforming CLFuzz (51.3%–54.8%), Cryptofuzz (43.8%–46.2%), Peach (72.3%–78.1%), and CDF (22.4%–25.1%). This improvement is attributed to Eidolon's noise-aware mutation strategy, which actively steers mutations away from noise overflow, thereby maximizing the proportion of valid test cases.

Answer to RQ2: The results show that Eidolon's noise-awareness improves valid input generation, thereby enhancing the overall effectiveness of fuzzing for FHE libraries.

Table 5. Valid input ratio across different fuzzers. Results are shown as Ratio (Valid/Total).

Library	Eidolon	CLFuzz	Cryptofuzz	CDF	Peach
OpenFHE	**89.2%** (8,375/9,389)	51.3% (7,108/13,856)	43.8% (5,683/12,975)	22.4% (3,524/15,732)	78.1% (5,815/7,446)
SEAL	**92.1%** (38,864/42,198)	54.8% (28,569/52,134)	46.2% (23,043/49,876)	25.1% (14,111/56,218)	72.3% (19,874/27,488)
HElib	**90.5%** (20,598/22,760)	53.1% (15,110/28,456)	45.4% (12,173/26,813)	23.8% (7,436/31,245)	73.7% (10,857/14,731)

6.3 Comparison with Other Fuzzers

To evaluate the code coverage performance of Eidolon, we conducted 24-hour experiments on each tested FHE library using Eidolon, CLFuzz, Cryptofuzz, CDF and Peach. For comparison, we adopt LibFuzzer's 'ft': metric [32], which integrates multiple coverage signals (such as edge coverage, edge counters, value profiles, and indirect caller/callee pairs) into a unified measure.

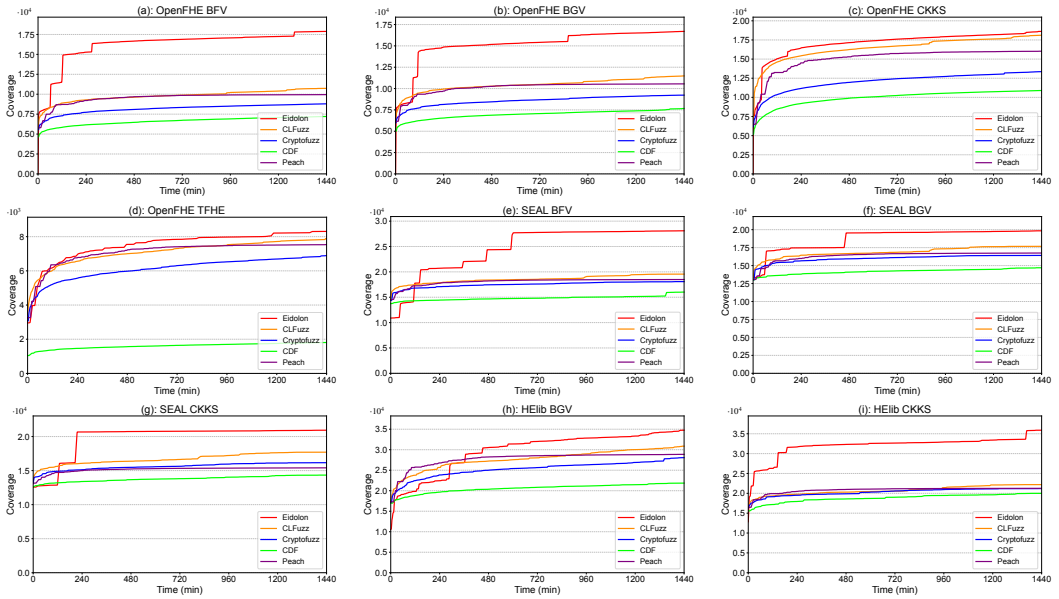

Fig. 9. Coverage trends of Eidolon, CLFuzz, Cryptofuzz, CDF and Peach over 24 hours.

To observe the trends of coverage growth over time, we record the coverage every minute over 24 hours. Figure 9 shows the growing trend of coverage achieved by different fuzzing tools during the 24-hour experiments.

We can observe that Eidolon achieves a rapid increase in coverage within the first 480 minutes (i.e., the initial 8 hours). After 24 hours, the coverage on SEAL converges across its three schemes, while the other libraries continue to show growth. In contrast, the baseline tools typically match Eidolon's coverage only during the first 30 minutes. As fuzzing continues, Eidolon surpasses them and maintains its advantage. This demonstrates that, over time, Eidolon leverages residual noise to construct increasingly deeper arithmetic expressions, thereby exploring more computational logic. After approximately one hour of execution, Eidolon already outperforms existing tools and continues to lead thereafter.

The detailed statistics are summarized in Table 6, which reports the final coverage values achieved by each tool across all tested FHE libraries. The results show that Eidolon covers 44,813, 62,852,

Table 6. Coverage achieved by different fuzzers across FHE libraries (24-hour experiments).

FHE Library	Tool	Coverage	Improvement	p-value	A_{12}
OpenFHE	Eidolon	61,504	-	-	-
	CLFuzz	48,221	27.5%	8.55e-11	0.9878
	Cryptofuzz	38,270	60.7%	3.87e-11	0.9967
	CDF	27,572	123.1%	3.02e-11	1.00
	Peach	44,060	39.6%	4.27e-11	0.9956
HElib	Eidolon	70,653	-	-	-
	CLFuzz	53,055	33.2%	1.54e-10	0.9811
	Cryptofuzz	49,254	43.4%	6.38e-11	0.9911
	CDF	41,850	68.8%	3.02e-11	1.00
	Peach	50,126	41.0%	8.55e-11	0.9878
SEAL	Eidolon	68,868	-	-	-
	CLFuzz	54,936	25.4%	7.75e-11	0.9889
	Cryptofuzz	50,649	36.0%	3.17e-11	0.9989
	CDF	45,042	52.9%	3.02e-11	1.00
	Peach	51,872	32.8%	5.73e-11	0.9923

86,561, and 54,967 more lines of code than CLFuzz, Cryptofuzz, CDF, and Peach, respectively. We employed the Vargha-Delaney A12 measure and the Mann-Whitney U test on the final coverage values from 30 independent repetitions. For every pairwise comparison against CLFuzz, Cryptofuzz, CDF, and Peach, the A12 measure is ≥ 0.714 and the p-value from the Mann-Whitney U test is ≤ 0.0001. These results confirm that Eidolon's coverage improvement is statistically significant. We attribute this improvement to two factors. First, by generating semantically valid arithmetic expressions (e.g., $x^3 + x + 1$) as inputs, Eidolon exercises the computation logic of FHE libraries that other tools do not reach. Second, through noise-awareness, Eidolon produces a higher proportion of valid inputs, enabling broader exploration of the computational space.

Answer to RQ3: The results show that Eidolon achieves higher code coverage compared to other cryptographic testing tools.

7 Discussion

False Positives. Although we applied methods to reduce false positives caused by noise overflow, Eidolon still reported two types of false positives. The first type arises from violations of implicit API preconditions. For example, in our tests on OpenFHE, Eidolon found that the EvalDivide operation produced incorrect results on certain negative input ranges. Developers clarified this was expected behavior due to the function's underlying approximation algorithm, which has limitations not enforced by explicit API checks.

The second type involves precision loss that appears as a correctness bug. We observed cases where simple additions yielded incorrect outputs. This was caused by an expected 'precision budget overflow' where internal scaling exceeded the modulus capacity. Developers noted that while this behavior is difficult to prevent, precision trade-offs are also sometimes utilized for efficient implementations in integer applications. These cases constitute approximately 7% of all potential bugs flagged, with a limited impact on the overall effectiveness of Eidolon. Once a false positive pattern is identified through developer feedback, it can be filtered from future reports.

Impact of Threshold Selection. To evaluate the impact of the τ_{low} threshold on Eidolon's performance, we conducted a sensitivity analysis on SEAL's BFV scheme over a 12-hour period, measuring the number of valid inputs generated. The results, shown in Table 7, indicate that the default setting of 10% achieved the best balance, allowing the generation of deep computations while

Table 7. Impact of the τ_{low} threshold on valid input generation for SEAL (BFV) over 12 hours.

τ_{low} Threshold	Valid Inputs Generated
3%	1970
5%	2068
10% (Default)	2217

maintaining validity. Although other thresholds were also viable, the 10% setting was consistently the most effective for exploring the computational space.

Future Work. Future directions include developing equivalence transformations aligned with FHE-specific operations like bootstrapping, and extending Eidolon to detect performance bugs such as unexpected noise consumption or bootstrapping instability.

Table 8. Ablation study of mutation strategies on OpenFHE BFV (24-hour experiments).

Strategy	**Valid Inputs**	**Coverage**
Default	2,387	40,412
Only High-Noise	7,102	45,205
Only Low-Noise	8,540	50,912
Eidolon	**9,435**	**61,529**

Impact of Mutator. We conducted an ablation study on OpenFHE BFV comparing four mutation strategies. As shown in Table 8, the Default strategy generates only 2,387 valid inputs with 40,412 coverage. Applying only the High-Noise or Low-Noise mutator improves both metrics, while Eidolon's combined strategy achieves the best results (9,435 valid inputs and 61,529 coverage). This confirms that the two mutators serve complementary roles. The High-Noise Mutator pushes toward deep computational boundaries, while the Low-Noise Mutator performs fine-grained exploration near those boundaries.

Table 9. Impact of the seed corpus on Eidolon's coverage and efficiency.

Variant	**Cov. (1h)**	**Cov. (24h)**	**Valid (1h)**	**Valid (24h)**	**First Bug**	**All Bugs**	**Convergence**
Eidolon-NoSeed	18.5%	57.6%	620	10,491	<0.1h	~26.2h	~28.8h
Eidolon	**28.3%**	**68.3%**	**1,250**	**13,834**	**<0.1h**	**~23.1h**	**~20.5h**

Impact of the Seed Corpus. We compared Eidolon against Eidolon-NoSeed, a variant without initial seeds. As shown in Table 9, the seed corpus provided significant coverage advantages. Eidolon achieved 53% higher coverage within the first hour (28.3% vs. 18.5%) and maintained an 18.6% lead at 24 hours (68.3% vs. 57.6%). The efficiency results further confirm this advantage. Eidolon generated 1,250 valid inputs in the first hour compared to Eidolon-NoSeed's 620, and reached coverage convergence at approximately 20.5 hours versus 28.8 hours. These results indicate that the seeds do not merely accelerate fuzzing but provide structurally diverse starting points that enable the noise-aware strategy to quickly reach deep computational regions.

8 RELATED WORK

8.1 Test Approaches for FHE

Current testing approaches for FHE libraries include static analysis tools like Klocwork [40] and dynamic methods like Project Wycheproof [24] and HEtest [48]. While valuable for conformance testing, these methods rely on fixed test datasets and cannot automatically generate diverse inputs. Fuzzing frameworks such as Cryptofuzz [49] and CDF [4] utilize differential testing across libraries, while CLFuzz [52] extends this with semantic-aware generation. Grammar-based fuzzers like Peach [35] can generate syntactically correct inputs based on predefined rules. However, these tools are primarily designed for testing cryptographic algorithms or generating syntactically valid inputs, rather than FHE computations, and do not incorporate noise-awareness to navigate the computational space defined by the noise budget. A separate line of work, such as HEDIFF [39], focuses on application-level robustness rather than library correctness.

8.2 Metamorphic Testing

Metamorphic testing (MT) [9] addresses the oracle problem by verifying Metamorphic Relations (MRs) across multiple inputs and their expected outputs. For instance, EMI [31] detects miscompilation bugs by generating semantically equivalent program variants, and MT has been applied to cybersecurity [10] for validating security-critical implementations. Eidolon adopts this paradigm by defining equivalence relations between different arithmetic forms (e.g., Standard vs. Horner). Unlike these prior works where transformations are generally unconstrained, Eidolon must incorporate noise-awareness to ensure that equivalent transformations remain computationally valid within the strict noise of FHE.

8.3 Fuzzing

Fuzzing is an effective technique for uncovering implementation bugs [34]. Generation-based tools such as Peach [35] construct inputs from grammars to ensure syntactic validity, while mutation-based tools such as AFL [23], AFL++[18], and LibFuzzer[32] use coverage-guided feedback to discover boundary conditions. Recent techniques combine specialized feedback signals or oracles in the fuzzing process. For instance, metamorphic fuzz testing [28] combines MT with fuzzing to detect faults in autonomous driving systems, QFuzz [36] uses execution costs for side-channel leakage quantification, and SQLancer [41] employs a containment oracle for DBMS logic bugs. Eidolon distinguishes itself by utilizing residual noise as feedback and Equivalence Expression Transformation as its oracle to detect bugs in the noise-constrained FHE computational space.

9 Conclusion

In this paper, we propose Eidolon, a noise-aware fuzzer for the vulnerability detection of FHE libraries. Eidolon first leverages noise as a feedback signal to guide the mutation of arithmetic expressions, enabling systematic exploration of deeper computational paths within the available noise. It then introduces an Equivalence Expression Transformation technique that generates mathematically equivalent but structurally diverse expressions, providing a reliable oracle to detect inconsistencies within a single library implementation.

We implemented and evaluated Eidolon on four widely used FHE libraries. Compared with existing cryptographic and grammar-based fuzzers CLFuzz, Cryptofuzz, CDF, and Peach, Eidolon achieves 28.7%, 45.5%, 75.6%, and 37.6% higher code coverage, respectively. Furthermore, Eidolon uncovered 20 previously unknown bugs, 10 of which have been fixed and 12 assigned CVE identifiers. Our future work will extend Eidolon to support more FHE libraries and enhance it by supporting more types of bugs.

10 Data Availability

We follow the FSE Open Science Policy and release our code and data on https://anonymous.4open.science/r/Jsd2wwewq0sdaSvsFds35nxsjai/

Acknowledgments

We sincerely thank the reviewers for their valuable suggestions. This work was supported by the National Cryptography Science Foundation of China under Grant 2025NCSF02055, and the National Key Research and Development Program of China under Grant 2024YFF1401303.

References

[1] Jonathan Anastasia and Derek Ho. 2024. *Stepping through the looking glass of privacy-enhancing technologies.* Mastercard. https://www.mastercard.com/news/perspectives/2024/stepping-through-the-looking-glass-of-privacy-enhancing-technologies/

[2] Apple. 2024. Combining Machine Learning and Homomorphic Encryption in the Apple Ecosystem. https://machinelearning.apple.com/research/homomorphic-encryption

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.

[4] Jean-Philippe Aumasson and Yolan Romailler. 2017. Automated testing of crypto software using differential fuzzing. *Black Hat USA* 7 (2017), 2017.

[5] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[6] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. 2020. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics* 13, Suppl 7 (2020), 83.

[7] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual cryptology conference*. Springer, 868–886.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[9] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. doi:10.1145/3143561

[10] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. 2016. Metamorphic Testing for Cybersecurity. *Computer* 49, 6 (June 2016), 48–55. doi:10.1109/MC.2016.176

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 360–384.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*. Springer, 409–437.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[15] Wikipedia contributors. 2025. Horner's method. https://en.wikipedia.org/wiki/Horner%27s_method Accessed: 2025-09-12.

[16] Ana Costache and Nigel P Smart. 2016. Which ring based somewhat homomorphic encryption scheme is best?. In *Cryptographers' Track at the RSA Conference*. Springer, 325–340.

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[19] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.

[20] Craig Gentry. 2010. Computing arbitrary functions of encrypted data. *Commun. ACM* 53, 3 (2010), 97–105.
[21] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer, 850–867.
[22] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual cryptology conference*. Springer, 75–92.
[23] Google. 2015. American Fuzzy Lop. https://github.com/google/AFL
[24] Google. 2019. Project Wycheproof. https://github.com/google/wycheproof.
[25] Google. 2021. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.
[26] Google. 2021. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[27] Shai Halevi and Victor Shoup. 2021. Bootstrapping for helib. *Journal of Cryptology* 34, 1 (2021), 7.
[28] Jia Cheng Han and Zhi Quan Zhou. 2020. Metamorphic Fuzz Testing of Autonomous Vehicles. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 380–385. doi:10.1145/3387940.3392252
[29] HElib. 2023. HElib release v2.3.0. https://github.com/homenc/HElib/releases/tag/v2.3.0
[30] homenc. 2025. HElib. https://github.com/homenc/HElib. Accessed: 2025-08-19.
[31] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334
[32] LLVM. 2025. LibFuzzer. https://llvm.org/docs/LibFuzzer.html Accessed: 2025-09-09.
[33] Microsoft. 2025. SymCrypt. https://github.com/microsoft/SymCrypt Accessed: 2025-09-11.
[34] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).
[35] Mozilla Security Team. 2025. Peach Fuzzer. https://github.com/MozillaSecurity/peach Accessed: 2025-09-09.
[36] Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: quantitative fuzzing for side channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 257–269. doi:10.1145/3460319.3464817
[37] OpenFHE. 2024. OpenFHE release v1.2.1. https://github.com/openfheorg/openfhe-development/releases/tag/v1.2.1
[38] OpenSSL. 2025. OpenSSL. https://github.com/openssl/openssl Accessed: 2025-09-11.
[39] Yiteng Peng, Daoyuan Wu, Zhibo Liu, Dongwei Xiao, Zhenlan Ji, Juergen Rahmel, and Shuai Wang. 2025. Testing and understanding deviation behaviors in fhe-hardened machine learning models. In *IEEE Computer Society*. 644–644.
[40] Perforce. 2025. Klocwork. https://www.perforce.com/products/klocwork
[41] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 38, 16 pages.
[42] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..
[43] SEAL. 2024. SEAL release v4.1.2. https://github.com/microsoft/SEAL/releases/tag/v4.1.2
[44] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.
[45] OpenFHE Development Team. 2025. Addition for CKKS FLEXIBLE* modes at multiplicative depth = 0 returns incorrect results for batch size = 1 or 2. https://github.com/openfheorg/openfhe-development/issues/728 Accessed: 2025-08-19.
[46] TFHE. 2017. TFHE release v1.0.1. https://github.com/tfhe/tfhe/releases/tag/v1.0.1, 2017.
[47] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[48] Mayank Varia, Sophia Yakoubov, and Yang Yang. 2015. HEtest: A homomorphic encryption testing framework. In *International Conference on Financial Cryptography and Data Security*. Springer, 213–230.
[49] Guido Vranken. 2025. Cryptofuzz. https://github.com/guidovranken/cryptofuzz Accessed: 2025-08-19.
[50] Wikipedia contributors. 2025. Homomorphic encryption. https://en.wikipedia.org/wiki/Homomorphic_encryption Accessed: 2025-08-19.
[51] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1309–1335.
[52] Yuanhang Zhou, Fuchen Ma, Yuanliang Chen, Meng Ren, and Yu Jiang. 2023. CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 45 (Dec. 2023), 28 pages. doi:10.1145/3628160

Received 2026-02-24; accepted 2026-03-24