# CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-Aware Fuzzing

YUANHANG ZHOU, Tsinghua University, China
FUCHEN MA, Tsinghua University, China
YUANLIANG CHEN, Tsinghua University, China
MENG REN, Tsinghua University, China
YU JIANG, Tsinghua University, China

Cryptography is a core component of many security applications, and flaws hidden in its implementation will affect the functional integrity or more severely, pose threats to data security. Hence, guaranteeing the correctness of the implementation is important. However, the semantic characteristics (e.g. diverse input data and complex functional transformation) challenge those traditional program validation techniques (e.g. static analysis and dynamic fuzzing). In this paper, we propose CLFuzz, a semantic-aware fuzzer for the vulnerability detection of cryptographic algorithm implementation. CLFuzz first extracts the semantic information of targeted algorithms including their cryptographic-specific constraints and function signatures. Based on them, CLFuzz generates high-quality input data adaptively to trigger error-prone situations efficiently. Furthermore, CLFuzz applies innovative logical cross-check that strengthens the logical bug detection ability. We evaluate CLFuzz on the widely-used implementations of 54 cryptographic algorithms. It outperforms state-of-the-art cryptographic fuzzing tools. For example, compared with Cryptofuzz, it achieves a coverage speedup of 3.4X and increases the final coverage by 14.4%. Furthermore, CLFuzz has detected 12 previously unknown implementation bugs in 8 cryptographic algorithms (e.g. CMAC in OpenSSL and Message Digest in SymCrypt), most of which are security-critical and have been successfully collected in the national vulnerability database (7 in NVD/CNVD) and is awarded by the Microsoft bounty program (2 for $1000).

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Cryptographic Algorithm, Fuzzing, Implementation Bug

## 1 INTRODUCTION

Cryptographic algorithms play a crucial role in securing online information transmission. To cater to the diverse requirements of different applications, various cryptographic algorithms have been proposed including message authentication codes [7], symmetric encryption and decryption [20], and elliptic curve cryptography [35], etc. They implement various functions and deal with different data structures. Existing implementations of cryptographic algorithms are diverse though they share common functions. Considering the significance of data privacy and the universality of cryptographic algorithm usage scenarios, bugs in the implementation can result in severe consequences.

Among the various approaches for vulnerability detection in cryptographic algorithm implementations, fuzzing stands out as a promising technique. However, most existing fuzzing approaches such as AFL, Libfuzzer, and their

Authors' addresses: Yuanhang Zhou, Tsinghua University, Beijing, China, zhouyh22@mails.tsinghua.edu.cn; Fuchen Ma, Tsinghua University, Beijing, China, mafc19@mails.tsinghua.edu.cn; Yuanliang Chen, Tsinghua University, Beijing, China, sard.chen@gmail.com; Meng Ren, Tsinghua University, Beijing, China, rm19@mails.tsinghua.edu.cn; Yu Jiang, Tsinghua University, Beijing, China, jiangyu198964@126.com.

1049-331X/2023/10-ART $15.00
https://doi.org/10.1145/3628160

derivatives are not suitable for testing cryptographic algorithms. These tools lack the necessary understanding of the input specifications specific to cryptographic algorithms and lack a test engine capable of triggering the execution of such implementations. But some attempts have been made for fuzzing the cryptographic implementations. For example, Cryptofuzz [32] is a differential fuzz testing tool based on Libfuzzer [42] that supports most mainstream algorithms and has discovered many bugs. It provides a self-defined custom generator and constructs seeds according to the input data fields of different algorithms. It performs differential testing on the results of different implementation versions of the same algorithm. CDF [5] is another differential fuzzing tool that detects implementation issues in cryptographic algorithms. CDF has been applied for several algorithms including pseudorandom functions (PRF) [64], symmetric encryption and decryption [20], Digital Signature Algorithms (DSA) [38] and ECDSA [37] functions. It also performs boundary testing using invalid parameters to trigger and detect common bugs.

These fuzzing methods have been widely adopted to improve the correctness of cryptographic algorithm implementations and have shown significant advancements. However, their specific requirements for input data and dependence on implementation versions pose certain challenges for conducting in-depth vulnerability detection.

The first challenge is that different categories of cryptographic algorithms have diverse semantic requirements for input data, making it difficult to generate all-purpose test inputs that can satisfy all these requirements without adaptive generation strategies. If the input requirements are not met, the validation phase will interrupt the execution, and the main logic of the algorithm cannot be triggered. For instance, when testing the Advanced Encryption Standard 128 (AES-128) [19] block cipher, which requires a key size of 128 bits, providing a key that is not 128 bits long will result in a failed data check and return an empty result. If this situation occurs repeatedly, it will lead to inefficient testing.

The second challenge lies in the complex functional transformations and data operations involved in the implementation of cryptographic algorithms. Traditional random input generation or coverage-guided mutation strategies struggle to trigger error-prone situations effectively. Boundary conditions, which arise from extreme lengths or special values of inputs, are examples of such scenarios. While these conditions may not frequently occur in everyday use, they have the potential to trigger abnormal behaviors in applications due to missing or incorrect processing. However, the input test data generated by traditional coverage-guided mutation strategies often fail to efficiently trigger such edge conditions.

The third challenge is that some cryptographic algorithms have limited implementations, making it difficult to perform effective differential testing for bug detection. For a certain algorithm, many calculation modes are available, each of which differs in the implementation. It is almost impossible for every implementation to cover all these modes. For example, there are over a hundred block cipher algorithms [22] supported for symmetric encryption and decryption. OpenSSL [51] implements over 140 of them while wolfCrypt [66] only supports about 50. As a result, when fuzzing the modes that only have a few implementations, the number of results is insufficient for differential testing, leading to false negatives in bug detection.

To address the aforementioned challenges, we propose CLFuzz, a semantic-aware fuzzer for the vulnerability detection of cryptographic algorithms' implementations. It takes full account of the characteristics of cryptographic algorithms to conduct adaptive test input generation and bug detection. First, CLFuzz extracts the semantic information including the cryptographic-specific constraints and function signatures of each algorithm. Then, CLFuzz implements adaptive test input construction strategies for the input fields of different data structures to improve the probability of triggering boundary situations. Moreover, for bug detection, besides the differential testing and runtime monitoring, CLFuzz presents strengthened logical cross-check across multiple test rounds based on the logical relationships among algorithms, which enables a more comprehensive and efficient vulnerability detection.

We evaluate CLFuzz on the widely-used implementations of 54 cryptographic algorithms in the aspects of bug detection and code coverage. It outperforms state-of-the-art tools Cryptofuzz and CDF. Specifically, compared to Cryptofuzz, it achieves a coverage speedup of 3.4X and increases the final coverage by 14.4%. Compared to CDF, the final coverage is increased by 538.5% and it takes CLFuzz little time (<1 minute) to reach the final coverage of CDF (>4 hours) in most cases. We also conducted an ablation study to show the contributions of CLFuzz's input generation and bug detection strategies. CLFuzz has detected 12 previously unknown implementation bugs in 8 algorithms (e.g. CMAC in OpenSSL and Digest in SymCrypt [46]), most of which are security-critical and have been successfully collected in the national vulnerability database (7 in NVD/CNVD) and awarded by the Microsoft bounty program (2 for $1000).

In summary, this paper makes the following contributions:

(1) We propose a semantic-aware fuzz testing approach for in-depth vulnerability detection of cryptographic algorithms' implementations.
(2) We design and implement CLFuzz on widely used implementations of cryptographic algorithms. CLFuzz generates high-quality test inputs adaptively based on the extracted cryptographic-specific constraints and function signatures, and detects more vulnerabilities through the logical cross-check.
(3) We make comprehensive evaluations on CLFuzz to demonstrate its effectiveness. CLFuzz outperforms state-of-the-art tools Cryptofuzz and CDF in terms of coverage, speed, and bug detection ability. It has detected many security-critical vulnerabilities and is open-sourced[1] for practical usage.

The rest of the paper is organized as follows. Section 2, introduces the background of generation-based fuzzing and cryptographic algorithms. Section 3 illustrates our motivation by an example and describes challenges faced by existing work. We formally introduce the design of CLFuzz in Section 4. Section 5 evaluates CLFuzz with state-of-the-art tools. We further discuss some features and limitations of CLFuzz in Section 6 and related works in Section 7. Section 8 makes a conclusion.

## 2 BACKGROUND

### 2.1 Generation Based Fuzzing

Fuzzing originated in 1990 when Miller et al. [47] applied a random test tool to evaluate the reliability of UNIX utilities. Nowadays, fuzzing has become a popular and promising dynamic testing method for finding implementation bugs in software. The core of fuzzing is sending a large volume of inputs continually in an attempt to make the program perform in a manner that was not intended, which could be a memory crash, extreme resource usage, wrong output results, etc. There are many academic surveys [41, 45] that give a unified summary of the current fuzzing world.

Generation-based fuzzing [48] is one of the major categories of fuzz testing. Typical examples include Peach [49] and Sulley [1], who produce test inputs based on formulated specifications or data models which describe the requirements of test cases. Today, researchers have combined a generation-based fuzzing framework with domain-specific grammar requirements and introduced grammar-based fuzzing, which addresses the problem of strict input formats in some scenarios. The grammar can be handwritten, or be mined through program feature analysis. Tools like ProFuzzer [69], CodeAlchemist [33], NAUTILUS [4] use highly-structured configurations to generate syntactically and semantically valid fuzz inputs. Besides, CSmith [67] and LangFuzz [36] use CFGs (control flow graph) to generate valid test inputs. Zest [55] enhances generation-based fuzzers with deterministic parametric generators to find semantic valid input. ISLa [62] leverages solvers like Z3 on the configuration provided by developers to solve semantic constraints and predicates over grammar structure. SLF [68] leverages the input checks of AFL [26] to dig for valid inputs. In general, the quality of the test input has a direct impact on the

[1]CLFuzz is available at: https://github.com/wuliguozi/CLFuzz

effectiveness of a fuzzer. To create a well-designed test input generation strategy, a generation-based fuzzer typically requires a significant amount of upfront work to explore the input specifications.

### 2.2 Cryptographic Algorithm

The implementations of cryptographic algorithms are usually encapsulated as libraries and provide API function calls to each of the supported features. Nowadays, cryptography has evolved from a domain used only by government and military agencies to one commonly used by developers worldwide. As a result, a number of proprietary and open-source cryptographic algorithm implementations have gradually emerged. They are widely used by individuals and enterprises as core components in security applications for conducting secure online transactions, communicating via secure email, and B2B (business-to-business) transactions [63]. Therefore, the security and robustness of cryptographic algorithm implementations are essential to ensure the security of applications.

Cryptographic algorithms are implemented in diverse programming languages including C, Go, JavaScript, etc. Different implementations of the same algorithm are diverse from each other though providing similar functions. For example, SSL and TLS algorithms are implemented by both OpenSSL [51] and WolfSSL [66]. They provide the utilities for network connection security. However, they use different data structures and different dependencies to implement the same algorithmic logic. It is difficult to ensure the correctness of all implementations, although the cryptographic algorithm itself is sound in theory.

## 3 MOTIVATION

### 3.1 Cryptographic Algorithm Implementation Bugs

Due to the diversity and continuous updates of cryptographic algorithms, it is difficult to avoid implementation bugs. Fig. 1 shows a vulnerability of the `EVP_DecryptUpdate()` algorithm in OpenSSL. It is a recent bug and has been assigned with a CVE id: CVE-2021-23840 [18]. Code from line 1 to line 16 sets the value of the variable `fix_len` to 1. The function `evp_EncryptDecryptUpdate()` at line 17 calculates the value of `*outl` according to the parameter `inl`. As shown in line 20, if the value of `fix_len` is 1, the value of `*outl` will be fixed. If the original value of `*outl` plus `b` exceeds `INT_MAX`, `*outl` will overflow and present a negative value at line 21 while the function still returns 1, which means the function executed successfully. This could cause applications to behave incorrectly or crash. Developers have fixed it by adding additional checks before calculation to raise an error as shown in line 7 to line 10.

### 3.2 Challenges to Detect Such Bugs

Triggering this bug is challenging for present tools. This function does block cipher based symmetric encryption or decryption based on a predefined configuration. Before the execution of the main algorithm, there are strict checks to ensure that input elements meet the syntax requirements. For the above example, the location of the bug is in function `EVP_DecryptUpdate()`. At the beginning of this function, there are detailed checks on the first parameter `ctx` because it is the core of the configuration including key, IV [40], etc. To form a qualified `ctx`, as shown in the official demo [52] provided by OpenSSL, users need to call `EVP_DecryptInit_ex()` to set up the configurations including key, IV, cipher mode, etc. The input structure of `EVP_DecryptInit_ex()` is shown in Fig. 2. In `EVP_DecryptInit_ex()`, if the length of `key` or `IV` is not consistent with the semantic restrictions of the cipher, a stack overflow runtime error will occur during initialization. This is abnormal and will interrupt the execution. Therefore, users must ensure that the lengths of the key and IV meet the specifications. To trigger the execution of the code in Fig. 1, a fuzzer must generate test inputs that conform to the semantic specifications of the cipher mode it uses.

```
 if (ctx->final_used) {
    if (((PTRDIFF_T)out == (PTRDIFF_T)in)
        || ossl_is_partially_overlapping(out, in, b)) {
        ERR_raise(ERR_LIB_EVP, EVP_R_PARTIALLY_OVERLAPPING);
        return 0;
    }
++    if ((inl & ~(b - 1)) > INT_MAX - b) {
++        ERR_raise(ERR_LIB_EVP, EVP_R_OUTPUT_WOULD_OVERFLOW);
++        return 0;
++    }
    memcpy(out, ctx->final, b);
    out += b;
    fix_len = 1;
} else{
    fix_len = 0;
}
if (!evp_EncryptDecryptUpdate(ctx, out, outl, in, inl))
    return 0;
...
if (fix_len)
    *outl += b;

return 1;
```

Fig. 1. An overflow error in OpenSSL and its repair. If the input length is close to the maximum permissible length for an integer, the variable **outl* will overflow.

```
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *cipher, ENGINE *impl,
                                   const unsigned char *key, const unsigned char *iv)
```

Fig. 2. The input structure of the function EVP_DecryptInit_ex. It requires 5 parameters including **ctx, *cipher, *impl, *key* and **iv*

.

For Cryptofuzz, it is challenging to trigger this bug because randomly generated keys and IVs can hardly meet the length requirements. Fig. 3 is an example input that Cryptofuzz generated. For cipher mode AES_192_CBC, the requested key length is 24 bytes, and the requested IV length is 16 bytes.

However, the input shown in Fig. 3 has a key with 4 bytes and an IV with 8 bytes. Therefore, the execution of this input will be interrupted during the data check, which makes it impossible to trigger the code segment with bugs. Based on Libfuzzer, Cryptofuzz inherits the coverage feedback mechanism that can retain valid inputs, and can go through a long learning phase to gradually learn the input specifications. However, its learning phase leads to inefficient testing, and the test efficiency after learning is still not as high as setting the standards in advance. Furthermore, the bug can only be triggered when the length of `ciphertext` (which is represented as `inl` in Fig. 1) is close to the maximum permissible length for an integer. It is also very hard for Cryptofuzz to generate inputs that meet this requirement because of the small probability that this condition can be randomly achieved.

```
1  Running:
2  operation name: SymmetricDecrypt
3  ciphertext: {0x58, 0xc9, 0x28, 0xe3, 0x86, 0x0c, 0x58, 0x82, 0x38, 0x2c, 0x64, 0xa8, 0xd5, 0x46, 0x33, 0x02}
        (16 bytes)
4  tag: {0x63, 0x59, 0x29, 0x0e, 0x84, 0x9c, 0x64, 0x44, 0x62, 0xfd, 0xcf, 0x6a, 0xa3, 0xe0, 0x8d, 0x68} (16
        bytes)
5  aad: {}
6  cipher IV: {0xa5, 0x35, 0x75, 0x17, 0xa2, 0xe4, 0x3e, 0x14} (8 bytes)
7  cipher key: {0x92, 0xa6, 0xdc, 0x55} (4 bytes)
8  cipher: AES_192_CBC
```

Fig. 3. A test input generated by Cryptofuzz. The length of the key and IV do not match the specifications, and the length of the ciphertext cannot trigger the error condition.

```
1  Running:
2  operation name: SymmetricDecrypt
3  ciphertext: {0xd2, 0xe7, 0xaf, 0xee, ... , 0xdf, 0x4c, 0xa9,} (INT_MAX-1 bytes)
4  tag: {0x63, 0x59, 0x29, 0x0e, 0x84, 0x9c, 0x64, 0x44, 0x62, 0xfd, 0xcf, 0x6a, 0xa3, 0xe0, 0x8d, 0x68} (16
        bytes)
5  aad: {}
6  cipher IV: {0xfd, 0x0f, 0xcb, 0x5e, 0xcb, 0x56, 0xd3, 0x1d, 0x69, 0x9a, 0xbe, 0x52, 0xb5, 0x83, 0xeb, 0x1e}
        (16 bytes)
7  cipher key: {0x6a, 0xde, 0xdd, 0x9a, 0x2f, 0xed, 0xbc, 0x86, 0x8a, 0xc3, 0x49, 0xae, 0x5f, 0x04, 0xe1, 0x36,
        0x44, 0x6f, 0xb9, 0xb5, 0xde, 0xe5, 0x0a, 0xa2} (24 bytes)
8  cipher: AES_192_CBC
```

Fig. 4. An expected input that can trigger the bug. The length of IV is 16 bytes and the length of the key is 24 bytes. The length of ciphertext is extremely long.

To effectively find such bugs, a fuzzer needs to overcome three challenges: (1) It should explore the semantic restrictions of each cryptographic algorithm to construct valid inputs for execution. Specifically, it should obtain the constraints of the parameters of targeted algorithms, and generate inputs that conform to the check. (2) It needs to generate inputs for boundary conditions that probably trigger errors based on edge situations of each data type. For cryptographic functions, the boundary condition is usually related to extreme values such as the maximum of integer, long integer, or zero. Fig. 4 shows an expected input that can trigger the bug above. As shown in line 6 and 7, the generated key length and IV length meet the syntax rules of the target calculation mode, so it can pass the data check. Besides, as shown in line 3, the length of the input ciphertext is close to the maximum integer, which is a typical boundary condition of plaintext and leads to abnormal behaviors. (3) It needs more bug detection oracles besides differential testing. Some algorithms only have a few implementations which are insufficient for differential testing. Thus, new oracles should be designed to detect logical bugs as shown in this example.

## 4 CLFUZZ DESIGN

CLFuzz aims to thoroughly test the implementations of cryptographic algorithms. Fig. 5 illustrates the overall framework of CLFuzz. It generates high-quality test inputs based on extracted semantic information and then employs various bug detection techniques including abnormality monitoring, differential testing, and logical cross-check to detect potential bugs. In this section, we present the details of the core design: semantic information extraction, test input generation, and bug detection.

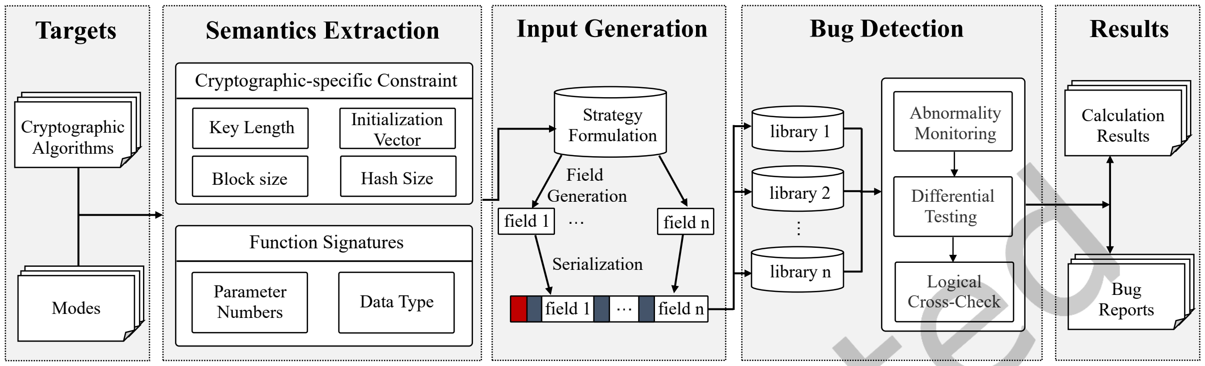


Fig. 5. The overall framework of CLFuzz. (1) CLFuzz takes in the targeted algorithms and modes. (2) CLFuzz conducts automatic semantic information extraction, extracting cryptographic-specific constraints and function signatures. (3) CLFuzz develops input generation strategies based on the semantic information, constructing each field of the test input and then serializing them into a byte string. (4) The test input triggers the execution. CLFuzz collects the results for abnormality monitoring, differential testing, and logical cross-checking. (5) CLFuzz outputs the results and generates a bug report.

## 4.1 Semantic Information Extraction

CLFuzz performs semantic information extraction to obtain various specifications regarding the input of the targeted cryptographic algorithms. Therefore, CLFuzz can generate high-quality test inputs that are efficient enough to reach the execution of the core process of algorithms and effective enough to trigger error-prone situations. CLFuzz focuses on two types of semantic information: *cryptographic-specific constraints* and *function signatures*. In the following, we detail the extraction approaches for them respectively.

*4.1.1 Cryptographic-specific Constraints.* To ensure security and integrity, cryptographic algorithms apply strict validity checks on the input data before the core calculation process. Let *cryptographic-specific constraints* be rules and requirements that must be followed when using cryptographic algorithms. CLFuzz extracts these domain-specific constraints to generate high-quality test inputs that can pass the check and trigger deeper execution logic.

CLFuzz considers 4 typical cryptographic-specific constraints:

- **Key length**: The key length used in cryptographic functions should meet the required length to ensure the functionality of the algorithm. For example, when conducting symmetric encryption, multiple block ciphers can be applied, including AES, SM4, ARIA, etc. They hold different restrictions on the key length. AES and ARIA support 128 bits, 192 bits, or 256 bits based on the calculation modes, while SM4 only supports 128 bits.
- **Initialization vector (IV) length**: In some cryptographic algorithms, such as block ciphers operating in Cipher Block Chaining (CBC) mode, an initialization vector is required. The length of the IV is required to be equal to the block size, which varies for different cipher modes.
- **Block size**: The block size in cryptography refers to the fixed-size blocks of data that are processed by block ciphers. It directly determines the length of the output ciphertext of symmetric encryption based on

a block cipher. As a result, the input ciphertext for corresponding decryption algorithms must meet this length requirement.
- **Hash size**: The hash size in cryptography refers to the size of the output produced by a hash function. In algorithms based on hash functions, such as key derivation functions, the supported output length is usually limited by the size of the underlying hash function.

CLFuzz covers the testing for 144 block ciphers and 18 hash functions, which hold various constraints mentioned above. For key length, IV length, and block size, CLFuzz leverages the public APIs provided by cryptographic libraries that enable users to query these cryptographic-specific constraints automatically. Since the functionality of a particular algorithm remains consistent across libraries, we take OpenSSL, which supports the most complete algorithms, as the source for extraction. Fig. 6 shows the APIs that take in the block cipher type and return the corresponding constraints. For hash length without API, CLFuzz encrypts an arbitrary plaintext using all available hash functions and records the length of the output.

```
int EVP_CIPHER_get_key_length(const EVP_CIPHER *cipher);
int EVP_CIPHER_get_iv_length(const EVP_CIPHER *cipher);
int EVP_CIPHER_get_block_size(const EVP_CIPHER *cipher);
```

Fig. 6. APIs used for querying key length, IV length, and block size.

*4.1.2 Function Signatures.* Let *function signatures* be the characteristics of a function, including the number and data types of parameters. CLFuzz extracts these signatures of targeted algorithms to generate syntactically valid test inputs that cover a wide range of scenarios, while also intentionally introducing errors to test the code's error-handling capabilities.

We illustrate how CLFuzz extracts the function signatures by the example in Fig. 7, which is the OpenSSL's implementation of symmetric encryption. The encryption process consists of 3 sub-processes: `EVP_EncryptInit` initializes an encryption context for a symmetric encryption algorithm; `EVP_EncryptUpdate` is called multiple times to encrypt the input plaintext in a block-by-block manner and produce the corresponding ciphertext; `EVP_EncryptFinal` finalizes the encryption process by encrypting the remaining plaintext data and producing the remaining ciphertext.

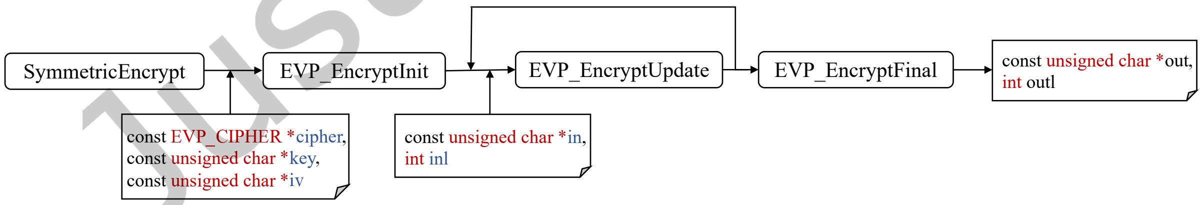


Fig. 7. The workflow of CLFuzz for extracting function signatures of the OpenSSL's symmetric encryption.

CLFuzz conducts function signature extraction on the declaration of targeted algorithms to obtain critical information about the input parameters required for the entire symmetric encryption process.

In this example, there are 5 parameters that are specified by test inputs, including `EVP_CIPHER *cipher`, `unsigned char *key`, `unsigned char *iv`, `unsigned char *in`, and `int inl`. CLFuzz mainly focuses on the parameter number and data types. The number of parameters guides CLFuzz to determine the appropriate length allocation for each parameter, ensuring that the total length of test input is within acceptable limits while also covering different argument combinations. The data types provide information about their syntax format and boundary conditions, including the minimum and maximum values they can accept and any special values that might trigger specific behavior in the function. By extracting these signatures, CLFuzz generates high-quality test cases that are comprehensive to cover a wide range of scenarios and are effective at identifying potential issues in the targeted algorithms.

## 4.2 Input Generation

In this section, we present how CLFuzz generates high-quality test inputs. CLFuzz conducts test input generation in a field-by-field method. CLFuzz first applies the selected strategy to form each field of the input, where each field represents a parameter of the targeted algorithm. These fields are then combined and serialized into a byte string, which is then passed to the test engine for execution.

CLFuzz establishes the generation strategies based on the extracted semantic information. For those input fields with cryptographic-specific significance, CLFuzz considers their extracted cryptographic-specific constraints to generate semantically valid inputs and ensure the test effectiveness. Additionally, CLFuzz applies chaotic strategies to generate a small portion of invalid inputs, specifically for testing the error-handling ability of the targeted implementations. For other fields with no cryptographic-specific constraints, CLFuzz selects the generation strategy based on their data type signatures. CLFuzz designs specific strategies for different data types considering their syntax format and boundary conditions. Overall, there are 6 strategies for field generation:

(1) Semantically Valid: The field should conform to the cryptographic-specific constraints.
(2) Extremely Small: The field reaches the minimum values that it can accept. For a string, the length should be (close to) 0. For an unsigned integer, its value should be (close to) 0.
(3) Extremely Large: The field reaches the maximum values that it can accept. The maximum accepted length or integer value is based on the configuration or execution platform.
(4) Special Content: The content of the field consists of extreme values of an unsigned byte such as 0x00 and 0xff.
(5) Empty Value: The input field is empty or undefined, indicating the absence of a value, such as a NULL.
(6) Random: The length and content of the input field are randomly generated.

After extracting all the function signatures of the targeted cryptographic algorithms, CLFuzz classifies the required input fields into 6 categories according to their roles and data types:

(1) Plaintext: Plaintext represents text that needs to be encrypted. In CLFuzz, it is displayed as a byte string of a certain length.
(2) Ciphertext: Ciphertext represents the output ciphertext of encryption. In turn, it is also an input field of decryption algorithms. In CLFuzz, it is displayed as a byte string of a certain length. Different from plaintext, the length of the ciphertext is strongly related to the encryption algorithm.
(3) Key: Keys in cryptographic algorithms. In CLFuzz, it is displayed as a byte string of a certain length. Keys always have two types: public keys and private keys. They hold different functions in algorithms but are similar in representation at the data level.
(4) IV: Initialization Vector. IV is widely used in the block cipher. Various block cipher modes include CBC (Cipher-block chaining) [23], Cipher Feedback (CFB) [22], Output Feedback (OFB) [22], and or so require an initial vector to participate in the encryption of the first block. In CLFuzz, it is displayed as a byte string of a certain length.
(5) Number: Mathematical numbers are widely used in cryptographic algorithms. For instance, a private key is a big number in elliptic curve algorithms. Therefore, big numbers often participate in the operation of mathematic calculations in cryptographic algorithms. Some cryptographic algorithms also provide

interfaces for basic mathematical operations of big numbers. In CLFuzz's test inputs, it is displayed as a string and will be converted into an integer during execution.

(6) Modes: A certain type of cryptographic algorithm may have multiple calculation modes. For example, Message-Digest Algorithm supports a variety of hash calculation modes such as MD5, SHA256, SM3, etc. Besides, elliptic curve algorithms have many kinds of curves, and symmetric encryption supports a variety of block cipher modes. Each test input aims at a specific mode of an algorithm.

For each category, CLFuzz customizes its generation strategy to better target the specific features for high test efficiency and thoroughly test the code's error-handling capabilities at the same time. Table 1 shows the generation strategies for each input field.

Table 1. Generation strategies for 6 types of the input field.

| Input Field | Semantically Valid | Extremely Small | Extremely Large | Special Content | Empty Value | Random |
|---|---|---|---|---|---|---|
| Plaintext | | ✓ | ✓ | ✓ | ✓ | ✓ |
| Ciphertext | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Key | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| IV | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Number | | ✓ | ✓ | | ✓ | ✓ |
| Modes | ✓ | | | | ✓ | |

For Plaintext and Number, there are typically no cryptographic semantic constraints on their length or content. However, boundary conditions of their lengths or contents may trigger bugs in extreme cases, making it important to apply corresponding generation strategies to these inputs. For Key, IV, and Ciphertext, cryptographic algorithms often hold strict semantic constraints on their length, making it essential to cover their specific features during test input generation. Additionally, some extreme values may trigger abnormal behavior, so these values are also included. Modes, on the other hand, are only valid if they are supported by the targeted implementations. Most modes are selected based on their semantic constraints, and in some cases, empty values are applied to cover extreme scenarios. For all other input fields, the random strategy is retained to cover general situations.

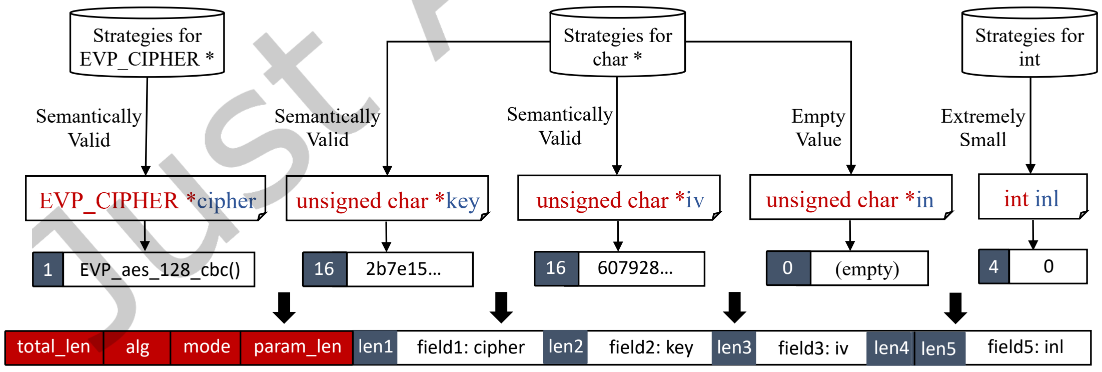


Fig. 8. The test input generation process of CLFuzz. This input targets symmetric encryption based on AES_128_CBC block cipher.

Fig. 8 illustrates how CLFuzz creates the test input and provides a detailed breakdown of its composition. The process begins with selecting the targeted algorithm and its calculation mode (if applicable) and extracting its semantic information. For instance, in this example, CLFuzz tests for symmetric encryption based on AES_128_CBC, whose semantic information has been extracted as shown in Fig. 7. Next, for each parameter, CLFuzz constructs its content using specific generation strategies, taking into account the constraints and boundary conditions extracted from its semantic information. For `cipher`, `key` and `iv`, CLFuzz applies the semantically valid strategy; for `in` and `inl`, CLFuzz applies empty value and extremely small strategy to test for edge conditions. To facilitate the subsequent field-splitting process, the length of each field is attached in front of its content. Finally, CLFuzz combines all generated fields into a serialized byte string and adds identifications, including the total length, targeted algorithm, and mode, to complete the construction of the test input. The resulting input is then delivered to the fuzz engine for execution on every implementation that supports the targeted algorithm.

## 4.3 Bug Detection

CLFuzz conducts a comprehensive three-stage bug detection process. To detect logical vulnerabilities, CLFuzz customizes specific oracles to conduct logical cross-check, which verifies the functional integrity and robustness of targeted algorithms. Besides, during the execution, CLFuzz closely monitors runtime indicators such as resource usage and execution time to detect any abnormal behaviors. After execution, CLFuzz collects the output results for differential testing to check the correctness of the execution outputs. We detail the three bug detection approaches in the following.

*Logical Cross-Check.* Cryptographic algorithms are built on logical properties among their functions, providing CLFuzz with a new approach of conducting logical cross-checks through multiple test rounds among different algorithms. Utilizing the domain-specific relationships among different algorithms, CLFuzz utilizes corresponding oracles to check if the cryptographic functions are working as expected and producing valid results.

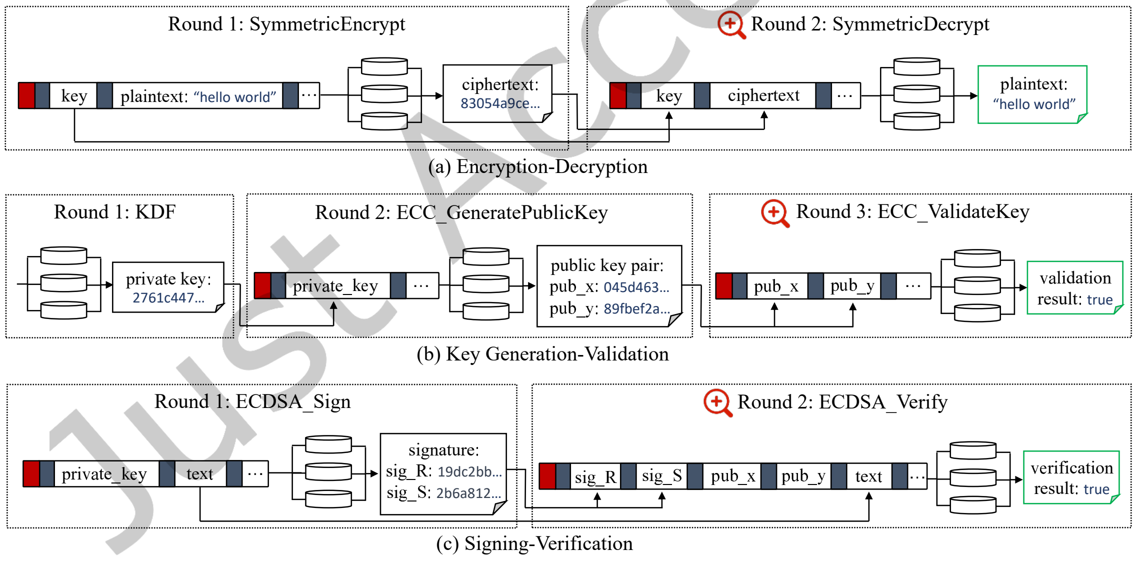


Fig. 9. An example of the logical cross-check among multiple test rounds. CLFuzz recycles the output of previous test rounds as input for subsequent tests.

Specifically, CLFuzz introduces the following oracles:

(1) Encryption-Decryption: CLFuzz tests symmetric encryption and decryption functions by generating plaintext input data, encrypting it using a given algorithm and key, and then attempting to decrypt the

resulting ciphertext using the same key and algorithm. This oracle verifies that the decrypted data matches the original plaintext input, thus ensuring the correctness of the encryption and decryption functions.

(2) Key Generation-Validation: Key generation is the process of creating a new key, while key validation involves checking its validity and using the key to perform encryption or decryption. CLFuzz takes the outputs of Key Derivation Functions (KDF) as the input for key validation to ensure that the generated keys are securely random and can be used to correctly encrypt and decrypt data.

(3) Signing-Verification: Digital signing uses a private key to generate a digital signature for a given input. Verification involves using the corresponding public key to check the authenticity of the signature. CLFuzz generates input data and signs it with a private key. It then verifies the signature using the corresponding public key to ensure that the data has not been tampered with, and that the generated signature is valid.

Fig. 9 shows examples of how CLFuzz conducts logical cross-check among multiple test rounds. For Encryption-Decryption in (a), CLFuzz records the specific configuration, including the key and cipher mode used for `SymmetricEncrypt`. It then applies the same configuration for the `SymmetricDecrypt` function and compares the decrypted results with the input plaintext. If the results match, the logical cross-check succeeds. For Key Generation-Validation in (b), in test round 1, CLFuzz executes KDF to generate a private key. Then in round 2, CLFuzz applies the private key as the input of `ECC_GeneratePublicKey` to obtain the public key pair. Finally, in round 3, CLFuzz takes the public key pair as the test input of `ECC_ValidateKey` function and checks the validation results. For Signing-Verification in (c), CLFuzz first conducts the same process as in round 1 and 2 in (b) to generate a key pair for `ECDSA_Sign`. Then, CLFuzz uses the private key to sign the message text and, in turn, uses the corresponding public key pair to verify the signatures using the `ECDSA_Verify` function. If the verification passes, the check succeeds. The logical cross-check enables uncovering hidden defects in the functionality of cryptographic implementations such as generating invalid keys, unverifiable signatures, etc.

To conduct cross-checks, CLFuzz uses an oracle recycling pool model to recycle the output results of previous test rounds and structure them into specified data formats. These formats are defined by several structs that contain the fields and key configurations of a particular dataset. For instance, to recycle a generated public key pair for elliptic curve algorithms, the data structure should include the following fields: 1) Curve ID, an ID that specifies the type of curve; 2) Private Key, the value of the private key used to generate this public key; and 3) Public Key, the value of the public key pair. Data of the same structure are collected in a pool for retention. To ensure the full utilization and continuous updating of the collected data, CLFuzz adopts a position cyclic data addition and extraction to maintain the pools. It uses position pointers to mark the position of the next addition and extraction, which move to the next position after each operation. This process ensures the validity of extracted data and enables CLFuzz to make full use of every piece of data.

Since the oracle recycling pool model can temporarily store the results of former test rounds, CLFuzz can selectively retrieve and utilize stored data from the pool in any subsequent rounds to generate new test inputs. In this way, CLFuzz does not need to consider the multiple execution sequences required by the oracles, but only focuses solely on the current round of testing. For instance, when utilizing the Signing-Verification oracle, CLFuzz executes the first round to generate a signature, but it doesn't immediately verify the signature. Instead, it stores the relevant parameters and results into the pool. When CLFuzz decides to fuzz the function responsible for signature verification in the future, it retrieves a signature from the pool and for generating new inputs, disregarding when the first round was executed. This approach eliminates the need for strict continuity between multiple rounds of logical cross-check. Therefore, when incorporating new implementations of oracles (e.g. new key generation-validation oracles for new kdf modes), CLFuzz doesn't require the implementation of the entire execution chain again. It can simply reuse the existing pools, allowing for efficient integration of additional oracles without significant overhead.

Besides, CLFuzz can validate the intermediate results, and conduct intensive mutations on them, allowing for the implementation of more variants for oracles. Firstly, CLFuzz performs abnormality monitoring and differential checking on the intermediate results obtained during multiple executions. Moreover, when reusing recycled data from the pool, CLFuzz provides flexibility in how it utilizes the data. It can either use the original recycled data as is or intentionally introduce modifications to the data during the input generation process. This allows CLFuzz to explore different variations and scenarios to test the robustness of the targeted algorithms. For example, when generating keys from a key derivation function, CLFuzz assumes that the originally generated keys are valid and should successfully pass subsequent validation checks. If the validation fails, CLFuzz identifies it as a potential vulnerability. On the other hand, if a modified input, or "polluted input," still manages to pass the validation, CLFuzz recognizes the failure in identifying invalidity. This enables CLFuzz to thoroughly test the error identification and handling capabilities of the targeted algorithms, probing for potential weaknesses or vulnerabilities.

*Runtime Monitoring.* To monitor the execution behavior of cryptographic implementations and detect abnormal runtime activity, CLFuzz instruments sanitizers, including AddressSanitizer [28] and UndefinedBehaviorSanitizer [30]. These sanitizers provide real-time feedback on the program's abnormal behavior and security posture. Working with the test inputs that trigger the extreme cases, CLFuzz captures runtime errors such as abnormal resource consumption, crashes, abnormal terminations, and memory corruptions, giving a comprehensive assessment of the reliability and robustness of the tested cryptographic implementations.

*Differential Testing.* For every test round, once the test inputs have been executed, CLFuzz collects the output results of all the targeted cryptographic implementations and applies differential checks. By comparing the results, any outputs that differ from the others are identified as potential bugs. This methodology is based on the fact that the function of a particular cryptographic algorithm remains consistent among various implementations. Therefore, for the same test input, the output results should be highly consistent. For test inputs that are supported by many tested implementations, the number of outputs is usually sufficient to highlight any abnormal results. However, if there are too few output results to determine which result is abnormal, further analysis is needed. In such cases, the results can be analyzed manually through the bug report.

## 5 EVALUATION

To illustrate CLFuzz's effectiveness, we proposed three research questions:

- **RQ1:** Can CLFuzz discover implementation bugs of cryptographic algorithms effectively?
- **RQ2:** Can CLFuzz achieve higher coverage and accelerate the growth of coverage compared with other tools?
- **RQ3:** Can CLFuzz boost the test input validity ratio compared with other tools?

To address these research questions, we designed the following experiments to compare the test effectiveness of CLFuzz and Cryptofuzz on 54 target cryptographic algorithms. For the various implementations under test, we provide unified test interfaces to trigger their executions. We use CLFuzz, Cryptofuzz, and CDF to fuzz these test interfaces and evaluate the execution results. All the experiments are executed for 20 hours and conducted 20 times on the same server with 256 GiB of memory and running 64-bit Ubuntu 20.04.2 LTS. To detect memory errors, we instrument the program under test with AddressSanitizer [28] and UndefinedBehaviorSanitizer [30] during the compilation of algorithms.

For better illustrations, we divide all the target algorithms into four categories and demonstrate the grouped results: Hash and Symmetric Function [20, 60], Elliptic Curve Algorithm [35], BLS Signatures [6], and Key Derivation Function [12]. All four categories are recognized classes of cryptographic algorithms that implement certain generalized functions.

(1) Hash and Symmetric Function: Hash function is a one-way function that maps data of an arbitrary size to a bit array of a fixed size. The symmetric function uses the same keys for both the encryption of plaintext and the decryption of ciphertext. This category contains the following algorithms:

```
Digest, HMAC, CMAC, SymmetricEncrypt, SymmetricDecrypt
```

(2) Elliptic Curve Algorithm: Elliptic curve cryptography is a public-key cryptography technique based on the algebraic structure of elliptic curves over finite fields. This category contains:

```
ECC_PrivateToPublic, ECC_ValidatePubkey, ECDH_Derive,
ECDSA_Sign, ECGDSA_Sign, ECRDSA_Sign, Schnorr_Sign,
ECDSA_Verify, ECGDSA_Verify, ECRDSA_Verify, Schnorr_Verify,
ECDSA_Recover, ECC_GenerateKeyPair, ECIES_Encrypt,
ECIES_Decrypt, ECC_Point_Add, ECC_Point_Mul
```

(3) BLS Signatures: BLS signature is a cryptographic signature scheme that uses bilinear pairing as well as elliptic curve. This category includes:

```
BLS_PrivateToPublic, BLS_PrivateToPublic_G2, BLS_Sign,
BLS_Verify, BLS_IsG1OnCurve, BLS_IsG2OnCurve,
BLS_GenerateKeyPair, BLS_Decompress_G1, BLS_Compress_G1,
BLS_Decompress_G2, BLS_Compress_G2, BLS_HashToG1, BLS_HashToG2,
BLS_Pairing, BLS_G1_Add, BLS_G1_Mul, BLS_G1_IsEq, BLS_G1_Neg,
BLS_G2_Add, BLS_G2_Mul, BLS_G2_IsEq, BLS_G2_Neg
```

(4) Key Derivation Function: Key derivation function derives keys from a secret value using pseudorandom function. This category contains:

```
KDF_SCRYPT, KDF_HKDF, KDF_TLS1_PRF, KDF_PBKDF, KDF_PBKDF1,
KDF_PBKDF2, KDF_ARGON2, KDF_SSH, KDF_X963, KDF_SP_800_108
```

## 5.1 Bugs in Cryptographic Implementations

CLFuzz has detected 12 previously unknown implementation bugs in 8 cryptographic algorithms (e.g. CMAC in OpenSSL and Message Digest in SymCrypt), most of which are security-critical and have been successfully collected in the national vulnerability database (7 in NVD/CNVD) and awarded by the Microsoft bounty program (2 for $1000).

The details of these bugs are listed in Table 2. These bugs contain multiple error types. 6 (#1, #3, #8, #9, #10, #11) of them involve logic flaws of code implementation, which means that the behaviors of the algorithm are inconsistent with the expected ones such as taking in invalid inputs or output wrong results. Another 4 (#2, #4, #5, #12) of these bugs are runtime errors caused by the imperfect implementation of some details including signed integer overflow and undefined behavior. The remaining 2 (#6, #7) are other security vulnerabilities that result in time leak [9] or denial of service.

Strategies for generating the test inputs that trigger these bugs and the bug detection strategy for them are listed in Table 3. Of all the vulnerabilities, 10 (#2, #4, #5, #6, #7, #8, #9, #10, #11, #12) cannot be detected by either Cryptofuzz or CDF. All of them involve semantic constraints and boundary cases that Cryptofuzz and CDF cannot generate inputs that can trigger them. Some are logical errors that require cryptographic-specific logical oracles to be discovered (#8, #9, #10). CLFuzz introduces logical cross-check that can detect them, while Cryptofuzz and CDF lack related bug detection strategies so they cannot find these bugs. For the rest 2 bugs (#1, #3), CLFuzz can detect them faster than Cryptofuzz and CDF.

Now we give a detailed illustration for Bug #4, #6, and #10 in Table 2 found by CLFuzz to see how implementation errors lead to abnormal behavior of the cryptographic function.

Table 2. CLFuzz has found 12 implementation bugs in 8 cryptographic algorithms.

| # | Library | Algorithm | Bug Type | Bug Description | ID | Bug Status |
|---|---|---|---|---|---|---|
| 1 | OpenSSL | CMAC | Logical Flaw | CMAC encryption result error. | CNVD-2021-86854 | confirmed |
| 2 | OpenSSL | SymmetricEncrypt | Integer Overflow | Symmetric encryption result error. | Bug#17869 | fixed |
| 3 | SymCrypt | Digest | Logical Flaw | MD4, MD5 encryption result error. | MSRC CASE #68154 | fixed |
| 4 | SymCrypt | CMAC | Integer Overflow | Signed integer overflow in CMAC-AES. | MSRC CASE #68154 | fixed |
| 5 | libtomcrypt | SymmetricEncrypt | Runtime Error | Undefined behavior in gcm_process | Bug#583 | fixed |
| 6 | mbed TLS | KDF_PBKDF | Denial of Service | Infinite loop in pkcs12 key generation. | CVE-2021-43666 | fixed |
| 7 | Crypto++ | ECC_PrivateToPublic | Time Leak [9] | Time leak in ecc public key generation. | CVE-2021-43398 | confirmed |
| 8 | chia-bls | BLS_PrivateToPublic | Logical Error | Invalid public key generated in BLS. | CNVD-2021-100406 | confirmed |
| 9 | sjcl | SymmetricEncrypt | Logical Flaw | Allowing dangerous IV in AES-CCM | CNVD-2021-88113 | reported |
| 10 | Crypto++ | ECC_PrivateToPublic | Logical Flaw | Invalid public key generated in ECC. | CNVD-2021-95295 | fixed |
| 11 | WolfCrypt | ECC_Point_Mul | Logical Flaw | Invalid multiplication result. | CNVD-2021-95292 | fixed |
| 12 | cifra | HMAC | Integer Overflow | Signed Integer Overflow in HMAC | Bug#20 | reported |

Table 3. Input generation and Bug detection strategies for detecting the bugs.

| ID | Input Generation Strategy | | Bug Detection Approach | | |
|---|---|---|---|---|---|
| | Cryptographic-specific Constraints | Function Signatures | Logical Cross-Check | Runtime Monitoring | Differential Testing |
| 1 | ✓ | | | | ✓ |
| 2 | ✓ | ✓ | | ✓ | |
| 3 | ✓ | | | | ✓ |
| 4 | ✓ | ✓ | | ✓ | |
| 5 | ✓ | ✓ | | ✓ | |
| 6 | ✓ | ✓ | | ✓ | |
| 7 | ✓ | ✓ | | ✓ | |
| 8 | ✓ | ✓ | ✓ | | |
| 9 | ✓ | | ✓ | | |
| 10 | ✓ | ✓ | ✓ | | |
| 11 | ✓ | ✓ | | | ✓ |
| 12 | ✓ | ✓ | | ✓ | |

*5.1.1 Case 1.* The first case is Bug #4 in Table 2. It was found in SymCrypt [46] and has been fixed in version 100.20.0. This vulnerability causes a signed integer overflow when calculating AES encryption and results in wrong results, which affects the normal function of this algorithm. It has been fixed by casting the values to unsigned integers.

The vulnerability lies in the main process of the AES encryption process, which can only be reached if the key length meets the semantic information of the supported cipher modes. Fig. 10 shows a test vector that can trigger this vulnerability. SymCrypt requires the key length to be a valid AES key (16, 24, or 32 bytes) and the *key* meets this restriction.

CLFuzz can detect this vulnerability in a short execution time. It has extracted the semantic information of AES cipher modes and learned the required length of the key. Therefore, the inputs generated by CLFuzz meet the requirements and can easily reach the main calculation process and trigger the bug. However, it would take a long time for the random generation strategy to random to the valid conditions.

```
void symcrypt_cmac_aes_128(){
    // key length: 16 bytes
    const uint8_t key[] = {0x9b, 0xe8, 0x42, 0x04, 0xa7, 0x1e, 0x31, 0xeb, 0xf4, 0xe0, 0xb1, 0x1a, 0xe2,
         0x5c, 0xac, 0x2f};
    const uint8_t msg[] = {0x9b, 0x6b, 0x6a, 0x5c, 0x1f, 0x0c, 0x5b, 0x7b, 0x01, 0x9b, 0x6b, 0x6a, 0x5c,
         0x1f, 0x0c, 0x5b, 0x7b} ;
    SYMCRYPT_AES_CMAC_EXPANDED_KEY xKey;
    uint8_t res[16] = {0};
    SymCryptAesCmacExpandKey( &xKey, key, sizeof(key));
    SymCryptAesCmac( &xKey, msg, sizeof( msg ), res );
}
```

Fig. 10. Code snippet that triggers Bug #4. The key length meets the semantic information, so Symcrypt overflows and gives incorrect encryption results.

*5.1.2 Case 2.* The second case is Bug #6 in Table 2. It was found in Mbed TLS [3] version 2.28.0 and earlier. This bug has been fixed in version 3.1.0. This vulnerability can cause an infinite loop and lead to a denial of service attack. The code snippet shown in Fig. 11 displays detailed information about the vulnerability.

```
1  static void pkcs12_fill_buffer( unsigned char *data, size_t data_len,
2      const unsigned char *filler, size_t fill_len )
3  {
4      unsigned char *p = data;
5      size_t use_len;
6      while( data_len > 0 ){
7          use_len = ( data_len > fill_len ) ? fill_len : data_len;
8          memcpy( p, filler, use_len );
9          p += use_len;
10         data_len -= use_len;
11     }
12 }
13 int mbedtls_pkcs12_derivation( unsigned char *data, size_t datalen,
14   const unsigned char *pwd, size_t pwdlen, ...)
15 {...
16     pkcs12_fill_buffer( pwd_block, v, pwd, pwdlen );
17 ...}
```

Fig. 11. Code snippet that causes an infinite loop. This vulnerability can be leveraged to conduct a denial-of-service attack.

`mbedtls_pkcs12_derivation()` is an exposed interface for key derivation function with pkcs12 [39] standard. Parameter pwd represents the input password pointer and `pwdlen` is its length. If `pwdlen` is 0, due to the lack of corresponding data check, mbed TLS executes normally to line 16 where `pkcs12_fill_buffer()` is called and passes a 0 for the fourth parameter: `fill_len`. At line 7, `use_len` is assigned a value of 0. Naturally, the value of `data_len` remains unchanged at line 10, causing an infinite loop. This vulnerability has been fixed by adding additional checks just after entering `mbedtls_pkcs12_derivation()` and before starting the loop in `pkcs12_fill_buffer()`.

To trigger this bug, a fuzzer has to generate a NULL or zero-length password for testing. If a random generation strategy is used to produce a random-length password within a certain length range, there is very little chance to trigger this bug. In contrast, CLFuzz considers the boundary condition of key length. Thus, CLFuzz greatly improves the probability of generating this particular input. So it can trigger this bug in a relatively short execution time.

```
void triggerBug(){
    using namespace CryptoPP;
    ECDSA<ECP, SHA256>::PrivateKey privateKey;
    ECDSA<ECP, SHA256>::PublicKey publicKey;
    string priv = "";
    const Integer privStr(priv.c_str());
    string ini = IntToString<>(privStr, 10);
    const DL_GroupParameters_EC<ECP>& cv = ASN1::brainpoolP256r1();
    // generate public key pair
    privateKey.Initialize(cv, privStr);
    privateKey.MakePublicKey(publicKey);
    AutoSeededRandomPool prng;
    // validation
    if(publicKey.Validate(prng, 3) == false)
    {   printf("invalid public key was generated!");}
}
```

Fig. 12. Code snippet that triggers Bug #10. Crypto++ generates an invalid public key pair without any exception prompt.

### 5.1.3 *Case 3.*

The third case is Bug #10 in Table 2. It is found in Crypto++ and is detected by logical cross-check. It happens when generating public key pairs from a private key string through ECDSA [37] in function `MakePublicKey()`. If the input private key string is empty, the generation will run normally. The output result is that $x$ and $y$ are both 0, which cannot pass the validation check function `Validate()` conducted by Crypto++ itself. In other words, Crypto++ generates an invalid public key pair without any exception prompt. Fig. 12 shows a test function that triggers this bug.

Tools like CryptoFuzz and CDF which apply differential testing cannot detect this bug. Once the output of algorithms is consistent, the result would be considered appropriate, ignoring its semantic correctness. In contrast, CLFuzz can detect it through logical cross-check. When fuzzing the function `MakePublicKey()`, the output result is collected and considered a valid public key pair. This key pair can be reused as the input of `Validate()` and the result is expected to be valid. For this case, an empty private key can be generated as the input of `MakePublicKey()` for it is a boundary condition and CLFuzz has taken full consideration of this situation. In turn, the output key pair of two 0 is recycled for the input of `Validate()` with an expected result of valid. However, the actual result is invalid. CLFuzz can detect this inconsistency and generate a bug report for subsequent analysis.

**Answer to RQ1:** CLFuzz is effective in detecting security-critical vulnerabilities in cryptographic algorithm implementations. In total, 12 vulnerabilities in 8 algorithms have been detected, where 7 are collected in national vulnerability databases and 2 are awarded by Microsoft bounty program.

## 5.2 Ablation Study

We conduct an ablation study to prove the effectiveness of CLFuzz's input generation and bug detection strategy separately.

First, we equip Cryptofuzz and CDF with CLFuzz's logical cross-check and runtime monitoring for 48 hours of execution to evaluate the bug-finding capabilities of their test input generation strategies only. Results show that Cryptofuzz and CDF still fail to find the 10 vulnerabilities due to the complex input requirements and unreachable boundary conditions they require. Even when provided with corresponding oracles, their pure random input generation strategies are unable to trigger the errors. In comparison, CLFuzz can detect all of them in under 6 hours. The results provide strong evidence that CLFuzz's input generation strategy possesses a stronger capability to trigger bugs.

Table 4. The time consumption of CLFuzz, Cryptofuzz and CDF on bug #1 and #3 that they all can detect. We collect the results by fuzzing all the functions as well as only fuzzing the Hash and Symmetric Function where the errors were found.

| Bug id | Fuzzing All the Targets | | | Fuzzing Hash and Symmetric Function | | |
|---|---|---|---|---|---|---|
| | CLFuzz | Cryptofuzz | CDF | CLFuzz | Cryptofuzz | CDF |
| #1 | 374.19s | >48h | >48h | 1.48s | 1,227.01s | 4,623.87s |
| #3 | 59.67s | 5,871.20s | 8,562.32s | 0.38s | 5.66s | 6.12s |

Secondly, we evaluate the time consumption of the three tools on bugs #1 and #3, which can be also detected by Cryptofuzz and CDF using their differential strategy. We run 20 rounds and take the average time cost. The results are shown in Table. 4. For bug #1, CLFuzz takes an average of 374.19 seconds to trigger it, while Cryptofuzz and CDF take more than 48 hours. If we manually specify the targeted function to the group of hash and symmetric function, it still takes Cryptofuzz 1,227.01 seconds and CDF 4,623.87 seconds to trigger the bug, while CLFuzz only needs 1.48 seconds. The significantly higher time cost is due to the slim chance of randomly generating input that conforms to its cryptographic-specific constraints on the test inputs. For bug #3, the average time required to detect it using CLFuzz, Cryptofuzz, and CDF is 59.67 seconds, 5,871.20 seconds, and 8,562.32 seconds respectively. They can detect bug #3 faster than bug #1 because the targeted function 'Digest' does not have complex semantic requirements on the length, or value of the input. However, CLFuzz can still detect it over 98 and 145 times faster than Cryptofuzz and CDF respectively. This is because CLFuzz is a generation-based fuzzer with high-quality test inputs that can cover the core logic of targeted algorithms in the early stage of testing, while others inherit the seed learning phase of traditional fuzzing, consuming considerable time breaking through the data check phase of a targeted algorithm that has already undergone testing.

These experiments demonstrate that CLFuzz not only detects more bugs than current approaches thanks to its logical cross-check mechanism but also outperforms them in detecting bugs that they all can detect, courtesy of its stronger input generation strategy.

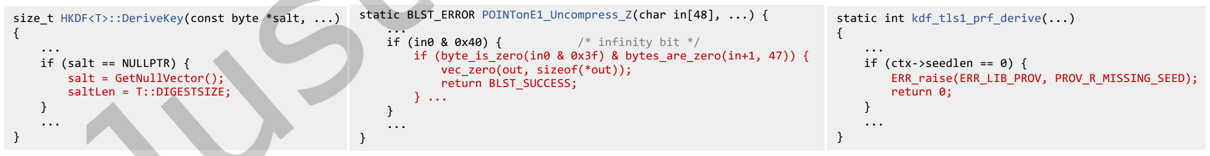
## 5.3 Code Coverage

To evaluate whether CLFuzz can achieve higher coverage in a shorter execution time compared with other fuzzing tools, we use the coverage metric 'ft:' [42]provided by Libfuzzer, which is a combination of coverage signals (edge coverage, edge counters, value profiles, indirect caller/callee pairs, etc.). This amalgamation of coverage features provides a holistic view of the achieved coverage and triggered behaviors. To ensure the validity and reliability of the experiment results, we conduct 20 rounds of repeated tests, each lasting 20 hours of execution time. We also apply statistical analysis to the coverage results. We apply Vargha-Delaney A12 measure [65] and the two-tailed Mann-Whitney U test [2] to assess the statistical significance of the differences in performance among the tested fuzzers. The Vargha-Delaney A12 measure is used to determine the effect size between two fuzzers being compared, which is reported as a value between 0 and 1. An $A_{12}$ value of 0.50 indicates no difference between the two fuzzers, while a value $\geq 0.714$ indicates significantly better performance, and a value of 1.0 indicates a 100% probability that one fuzzer outperforms the other. The Mann-Whitney U test is used to evaluate the null hypothesis significance, which tests whether there is a statistically significant difference in performance between the tested fuzzers. If the *p-value* is less than the pre-determined significance level (usually set to 0.05), we reject the null hypothesis and conclude that there is a statistically significant difference between the two groups.

Table 5. The statistical analysis of the coverage of Cryptofuzz, CDF, and CLFuzz on 4 categories of algorithms.

| Algorithm Category | Tool | Coverage | Improvement | *p-value* | $A_{12}$ |
|---|---|---|---|---|---|
| Hash & Symmetric Function | CLFuzz | 713,943 | - | - | - |
| | Cryptofuzz | 671,979 | 6.2% | 3.04e-124 | 0.7794 |
| | CDF | 126,279 | 565.0% | 0.0 | 1.0 |
| Elliptic Curve | CLFuzz | 336,836 | - | - | - |
| | Cryptofuzz | 296,277 | 13.7% | 2.20e-266 | 0.9110 |
| | CDF | 65,699 | 512.6% | 0.0 | 1.0 |
| BLS Signatures | CLFuzz | 105,442 | - | - | - |
| | Cryptofuzz | 79,371 | 32.8% | 7.27e-303 | 0.9384 |
| Key Derivation | CLFuzz | 373,004 | - | - | - |
| | Cryptofuzz | 356,161 | 4.7% | 2.62e-72 | 0.7519 |

The results are shown in Table 5. As it shows, for each category of cryptographic algorithms, CLFuzz achieves higher coverage than Cryptofuzz and CDF. For every pairwise comparison, the $A_{12}$ is $\geq 0.714$ and *p-value* is $\leq 0.05$, indicating that CLFuzz significantly outperforms Cryptofuzz and CDF in every category. The ratio of coverage improvement is influenced by the complexity of the cryptographic-specific constraints and the scale of implementation code for each category. In the case of Elliptic Curve and BLS Signatures, where there are more constraints on input fields, we observed a substantial increase in overall coverage. However, for Hash & Symmetric Functions and Key Derivation, where the constraints are relatively fewer, the increase in coverage was comparatively lower.



```
size_t HKDF<T>::DeriveKey(const byte *salt, ...)
{
    ...
    if (salt == NULLPTR) {
        salt = GetNullVector();
        saltLen = T::DIGESTSIZE;
    }
    ...
}
```

(a) Cryptopp: hkdf.h

```
static BLST_ERROR POINTonE1_Uncompress_Z(char in[48], ...) {
    ...
    if (in0 & 0x40) {           /* infinity bit */
        if (byte_is_zero(in0 & 0x3f) & bytes_are_zero(in+1, 47)) {
            vec_zero(out, sizeof(*out));
            return BLST_SUCCESS;
        } ...
    }
    ...
}
```

(b) BLST: e1.c

```
static int kdf_tls1_prf_derive(...)
{
    ...
    if (ctx->seedlen == 0) {
        ERR_raise(ERR_LIB_PROV, PROV_R_MISSING_SEED);
        return 0;
    }
    ...
}
```

(c) OpenSSL: tls1_prf.c

Fig. 13. Examples of the coverage difference between ÇLFuzz and others.

To evaluate the difference in actual code coverage between CLFuzz and existing approaches, we visualized their code line coverage and compared the differences in the lines of code covered. The results show that CLFuzz is capable of covering all the code lines that Cryptofuzz and CDF cover. Moreover, CLFuzz covers 24,006 and 57,793 more lines than Cryptofuzz and CDF respectively. These additional lines are covered only by CLFuzz due to two main reasons. Firstly, CLFuzz extracts semantic requirements that enable it to trigger all the core logic of the algorithms, while Cryptofuzz and CDF fail to pass data checks, resulting in many misses in the core process code. For example, of the file `camellia.cpp` [16] in library CryptoPP which implements symmetric encryption using the Camellia block cipher, CLFuzz is able to cover the entire implementation of this file, while Cryptofuzz and CDF fail to cover any line because they are unable to generate a valid input that meets the semantic requirements and trigger the encryption. Secondly, with the help of function signatures, CLFuzz covers more codes that deal with boundary conditions and error handling triggered by extreme and special values. Some examples are shown in Fig. 13, where red lines represent code that is only covered by CLFuzz. (a) [17] contains branches for handling extreme value `NULLPTR`. (b) [8] deals with special value. (c) [53] raises an exception for extreme value. CLFuzz leverages the extracted function signatures to trigger edge conditions and branches, resulting in better performance compared to Cryptofuzz and CDF.

To observe the trends of coverage growth over time, we record the coverage every minute over 20 hours, and we show the plots of coverage over the duration of each category in Fig. 14. The shadow represents the range of values in 20 repetitions of experiments, and the line represents the average value. At the beginning of the fuzz testing, the coverage grows rapidly for both CLFuzz and Cryptofuzz. When the execution time reaches 200 minutes, CLFuzz has achieves a significantly higher coverage than Cryptofuzz and CDF. In the follow-up phase, they both grow at a slow rate and maintain the gap. In addition, the fluctuation of the value is small, indicating that the optimization effect is stable.

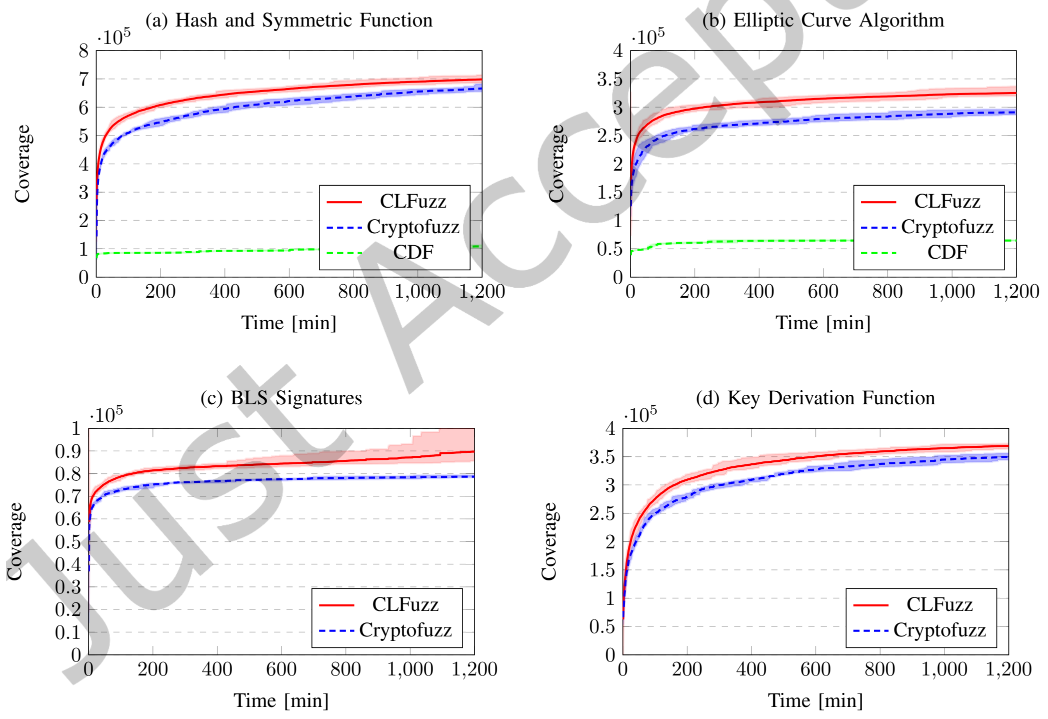


Fig. 14. Coverage trends of Cryptofuzz, CDF, and CLFuzz for 4 categories over 20 hours. CDF does not support the algorithms in the last two categories.

Comparing coverage statistics at the end of the campaign does not reveal the entire picture. CLFuzz significantly accelerates the coverage growth compared with Cryptofuzz and CDF in all categories. Table 6 shows the time

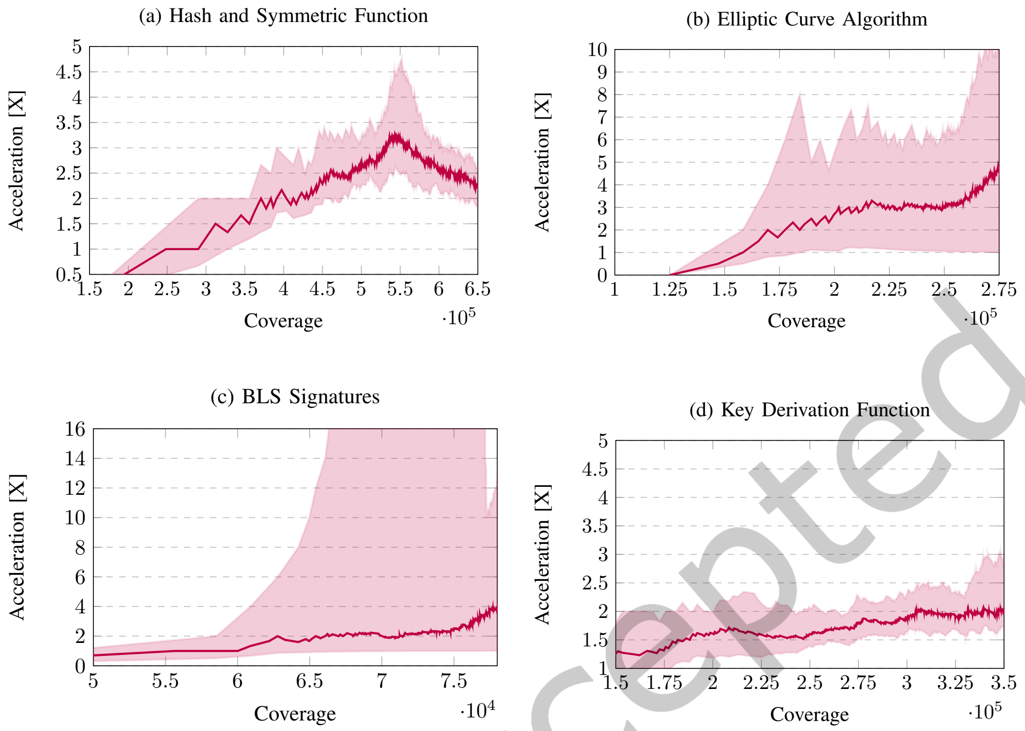


Fig. 15. Trend of CLFuzz over Cryptofuzz coverage growth acceleration on each algorithm category. For CDF, CLFuzz can reach its final within the first minute of testing, so we do not use graphs for comparison.

that CDF and CLFuzz used to achieve the same coverage as Cryptofuzz did over 20 hours. For CDF, we have been running it for 100 hours, and the results are still far from the target values. As for CLFuzz, for all four categories, it took 9.5, 4.3, 4.3, 10,0 hours to achieve the coverage of Cryptofuzz's 20-hour operation, with an acceleration of 2.1X, 4.6X, 4.7X, 2.0X respectively which has a great impact on the efficiency of fuzz testing.

Table 6. Time that CDF and CLFuzz use to achieve the top coverage of Cryptofuzz in 20h.

| Algorithm Category | Coverage | Time-Cryptofuzz[h] | Time-CDF[h] | Time-CLFuzz[h] | Acceleration Ratio |
|---|---|---|---|---|---|
| Hash & Symmetric Function | 671,979 | 20 | >100 | 9.5 | 2.1X/>10X |
| Elliptic Curve | 296,277 | 20 | >100 | 4.3 | 4.6X/>10X |
| BLS Signatures | 79,371 | 20 | - | 4.3 | 4.7X/- |
| Key Derivation | 356,161 | 20 | - | 10.0 | 2.0X/- |

More comprehensively, Fig.15 shows how many times faster CLFuzz is than Cryptofuzz when the code coverage reaches certain values of the horizontal coordinate. For CDF, CLFuzz can reach its final within the first minute of testing, so we do not use graphs for comparison. In the first half of the curve, the coverage increases extremely rapidly, resulting in an unstable acceleration. After entering the stage of stable coverage growth, the acceleration ratio rises steadily. Since the final coverage CLFuzz can reach is higher than Cryptofuzz, the value will finally tend to be infinity.

Table 7. The number of valid inputs of Cryptofuzz and CLFuzz on 4 classes of algorithms among 100000 generated inputs.

| Algorithm Category | SIC | VIC-Cryptofuzz | VIC-CLFuzz | Improvement |
|---|---|---|---|---|
| Hash & Symmetric Function | 100,000 | 80,721 | 92,976 | 15.2% |
| Elliptic Curve Algorithm | 100,000 | 62,736 | 85,894 | 36.9% |
| BLS Signatures | 100,000 | 55,700 | 96,366 | 73.0% |
| Key Derivation Function | 100,000 | 74,108 | 92,682 | 25.1% |

**Answer to RQ2:** The results show that CLFuzz can achieve a higher coverage and significantly accelerates the growth of coverage compared with state-of-the-art tools.

## 5.4 Test Input Validity

When the execution process is interrupted due to an invalid input field, an empty result will be returned. In this case, the testing cannot reach the main part of the algorithm, resulting in low test efficiency. On the contrary, if a test input can pass the data check and initialization stage and successfully reach the main process, we suppose that this input is valid to obtain significant outputs. To evaluate the improvement of the test input validity ratio, we calculate the following two indicators during the fuzzing process:

**SIC:** Supported Input Count. The total number of test inputs that are supported by the target cryptographic function.

**VIC:** Valid Input Count. The total number of test inputs that are valid enough to pass the data check and trigger the main process.

For CDF, the changeable part of inputs contains no configuration fields. The configurations are either defined in a fixed configuration file or provided by test files, both of which are predefined by users and have no relation to the inputs generated by CDF. As a result, the test range of CDF is quite narrow, and the input validity has no significance to CDF. Therefore, we only compare Cryptofuzz with CLFuzz in this evaluation.

We mark the time when **SIC** reaches $100,000$ and record the value of **VIC**. As shown in Table 7, for each categories, the proportion of valid inputs generated by CLFuzz is significantly high, reaching 93.0%, 85.9%, 96.4%, 92.3% respectively, and exceeding 15.2%, 36.9%, 73.0%, 25.1% compared to Cryptofuzz. Benefiting from the syntax and semantic sensitive generation strategies, CLFuzz produces high-validity inputs and increases the probability of triggering main algorithms. CLFuzz can then gain more valid outputs for adequate bug detection, and in turn, optimize the effectiveness of the whole fuzzing process. In addition, to detect bugs caused by incomplete data checks such as accepting unsupported input, CLFuzz still preserves the generation of a few invalid inputs. This strategy can be used to test the robustness of data checks provided by cryptographic algorithms themselves.

**Answer to RQ3:** The results show that CLFuzz can improve the validity ratio of generated test inputs, and then boosts the effectiveness of the test.

## 6 DISCUSSION

### 6.1 Extensibility on New Targets

Currently, CLFuzz has been applied to test the widely-used implementations of 54 cryptographic algorithms. To continuously improve its universality and integrity, we need to constantly supplement new algorithms or new implementations to enrich the tested elements. Therefore, extensibility is particularly important for its sustainable development. CLFuzz has already provided a standard workflow for new members. A class module has been defined for structured extensions as shown in Fig. 16.

```
class Module {
    public:
        const std::string name;
        const uint64_t ID;
        // Constructor
        Module(const std::string name) :
            // Some Initializations
        { }
        // virtual functions for algorithms
        virtual std::optional<component::Key> OpKDF_PBKDF(operation::KDF_PBKDF& op) {
            (void)op;
            return std::nullopt;
        }
        ...
}
```

Fig. 16. Structure of the class Module for testing new implementations of algorithms.

To add a new algorithm for testing, first, we need to implement the corresponding driver that conducts the functionality using the APIs provided by the targeted implementations. Fig. 17 shows an example of extending the current CLFuzz to the implementation of algorithm KDF_PBKDF in library yescrypt [54]. Firstly, we initialize a new module for yescrypt and declare the function KDF_PBKDF in line 1-5. Then, based on the documentation and test file, we obtain the process of how yescrypt conducts KDF_PBKDF and implement the driver as shown in line 7-26. We parse the test input generated by CLFuzz as the parameters for the APIs. In line 10-13, CLFuzz parses the input parameters including password, key length, salt, etc. Then in line 15-17, CLFuzz conducts the required semantic data check of yescrypt. Finally, in line 19-22, we execute the targeted API and get the output results, and return it in a uniform output format for subsequent checks.

Besides, we supplement the corresponding test input generation strategy with two steps. First, we extract new semantic information about this algorithm. Second, we explore its input fields and apply the specified generation strategies for different data types. After that, CLFuzz can support new targets. To add a new implementation to the test, we just instantiate a new instance of this class module and then overwrite the corresponding virtual function.

## 6.2 Completeness of Bug Detection

There is a great variety of cryptographic algorithms, and new algorithms or calculation modes are constantly coming out. As a result, the implementation of cryptographic algorithms is also continually updated. Different implementations support different algorithm ranges, some of which only focus on a certain category of algorithms. It is challenging to cover all the algorithms of the implementation completely during the test.

One main reason is that some algorithms or modes are only supported by a few implementations, resulting in an insufficient number of outputs for differential testing. Although CLFuzz has adopted logical cross-check that uses the logical properties of algorithms to enrich the approaches of bug detection, not all cryptographic algorithms have the semantic information that could be used. Another reason is that the levels of functional interfaces exposed by the different implementations are different from each other. For the convenience of users,

```
class yescrypt : public Module {
    public:
        yescrypt(void);
        std::optional<component::Key> yescrypt::OpKDF_PBKDF(operation::KDF_PBKDF& op) override;
};

std::optional<component::Key> yescrypt::OpKDF_PBKDF(operation::KDF_PBKDF& op) {
    std::optional<component::Key> ret = std::nullopt;
    // parse input parameters
    auto password = op.password.Get();
    unsigned int passwordLen = password.size();
    unsigned int keyLen = op.keySize;
    auto salt = op.salt.Get();
    // data check
    uint8_t out[64];
    assert(keyLen <= sizeof(out));
    assert(op.digestType == CF_DIGEST("SHA256"));
    // core funtionality
    PBKDF2_SHA256((const uint8_t *) password, passwordLen,
      (const uint8_t *) salt, saltlen, c, out, keyLen);
    // get output
    ret = component::Key(out, keyLen);
end:
    util::free(out);
    return ret;
}
```

Fig. 17. An example driver of function KDF_PBKDF in yescrypt.

some implementations only provide top-level interfaces, within which it has defined the execution process by itself. In contrast, some others provide more detailed interfaces. It is difficult to fully test the top-level and detailed interfaces at the same time.

## 7 RELATED WORK

### 7.1 Test Approaches for Cryptographic Algorithms

Currently, various testing approaches have been applied to cryptographic algorithms or APIs, involving static and dynamic tests. CryptoGuard [59] is a typical case for static vulnerability detection which focuses on cryptographic API misuse in massive-sized Java projects. For each common cryptographic vulnerability, CryptoGuard defines corresponding code rules of common cryptographic vulnerabilities to scan the projects. Furthermore, it refines program slices by identifying language-specific irrelevant elements such as common programming idioms and language restrictions to reduce false alerts. It has achieved objective results and has found multiple vulnerabilities in large-scale Apache projects and Android apps.

As for dynamic testing, there are also some existing achievements. Project Wycheproof [27] by Google, tests crypto algorithms against known attacks by a directed approach with tailored test vectors mindful of cryptographic theory and historic weaknesses. DifFuzz [50] can detect side-channel bugs by using resource-guided heuristics to find inputs that maximize the difference in resource consumption between two versions of the program. Another example is CDF [5], a dynamic test tool for cryptographic algorithms that combines fuzz testing and stateless dedicated test vectors of known vulnerabilities and edge cases of the tested algorithms. Besides, Cryptofuzz [32] is a differential fuzz testing project that analyzes cryptographic algorithms and compares their outputs to find implementation discrepancies. It can detect memory bugs with the aid of sanitizers [29]. Cryptofuzz supports a wide range of algorithms and has made remarkable achievements on cryptographic algorithm tests.

## 7.2 Traditional Fuzzing

Fuzzing is an effective and promising test method for detecting implementation bugs in algorithms. Traditional fuzzers can be divided into two types according to the seed production principle: generation-based fuzzers and mutation-based fuzzers [48].

Typical generation-based fuzzers includes Peach [49], Peach* [44], and Sulley [1]. They mainly focus on targets with strict input format requirements. They generate high-quality seeds for each round of testing based on format specifications predefined by users and guide further fuzzing processes according to the execution results.

Mutation-based fuzzers use initial seeds or previously produced seeds to generate new ones by making byte or bit-level changes to them. One of them is American Fuzzy Lop (AFL) [26], a coverage-guided fuzzer known for its exceptional performance on bug detection. To advance AFL's performance, a series of family tools [10, 24, 57, 58] has been developed. AFL++ [24] achieves faster speed, more and better mutations, and more and better instrumentation compared with AFL. AFLSmart [10] combines AFL's grey box fuzzing with high-level structural seeds to explore new input domains while maintaining seed validity. Besides, Libfuzzer [42] is an in-process, coverage-guided, evolutionary fuzzing engine. Under normal conditions, it unitizes diverse strategies for seed mutations. In addition, it also provides interfaces for users to customize mutation or generation strategies. By using this interface, Libfuzzer can produce structured seeds that meet the specifications of software under tests.

However, most of the existing traditional fuzzing works cannot be applied to cryptographic algorithms testing. The main reason is that they lack the knowledge of input specifications of cryptographic algorithms and the test engine that could trigger the execution of their implementation.

## 7.3 Property-based Testing

Besides the traditional bug detection approaches, there are also some current works that consider the semantic information of targeted algorithms during the bug detection stage. They design the properties that the programs should always satisfy, and use them as oracles to detect unexpected behaviors. This approach is known as property-based testing (PBT).

Property-based testing was popularized by the Haskell library QuickCheck [15]. QuickCheck requires the programmers to provide a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. FsCheck [25] uses the same approach to test for ṄET programs. Furthermore, some works optimize the random input generation strategy of PBT and present targeted property-based testing (TPBT). For example, Zest [55] extends QuickCheck's random-input generators into deterministic parametric generators to better explore the semantic analysis stage of test programs. Target [43] guides PBT with a search strategy. PropEr [56] integrate types and function specifications of Erlang with PBT. Besides, metamorphic testing (MT) is also a form of property-based testing. It is conducted based on Metamorphic Relations (MRs), which are necessary properties of the target function or algorithm in relation to multiple inputs and their expected outputs. It was initially proposed by TẎĊhen [13] in 1998 as a software verification technique, and has been widely applied to real-life applications for various types of software quality assessment. Surveys conducted by [14, 61] summarized the research results and application practices of MT. Some application examples include web service [11], imaging software testing [31], autonomous vehicles [34], and autonomous driving models [21]. The key advantage of PBT is that it can uncover logical implementation errors that may be missed by traditional testing methods. It can also help to identify and isolate defects, making it easier to debug and fix issues.

### 7.4 Main Difference

Unlike existing tools, CLFuzz stands out as a specialized generation-based fuzzer specifically designed for fuzzing cryptographic algorithms. It takes full advantage of the cryptographic-specific constraints to generate high-quality test inputs that effectively target the vulnerable components of cryptographic algorithm implementations. The inputs generated by CLFuzz not only satisfy the necessary data checks but also have the capability to trigger error-prone scenarios, thoroughly testing the resilience of these implementations.

Furthermore, CLFuzz leverages the distinctive logical relationships presented in cryptographic algorithms during the bug detection phase. Existing tools cannot address the challenges in cryptographic algorithm testing. CLFuzz first extracts and designs the logical relationships among 54 targeted algorithms and establishes the cryptographic-specific oracles that precisely cover the features of cryptographic algorithms. Then to combine the oracles with high-efficiency fuzzing, CLFuzz utilizes the oracle recycling pool model for temporarily storing the data of former test rounds. This eliminates the need for strict continuity between multiple rounds of logical cross-check, and therefore alleviates the burden of equipping different oracles, enhancing its extendability. Besides, CLFuzz can validate the intermediate results, and conduct intensive mutations on them, allowing for the implementation of more variants for oracles.

## 8 CONCLUSION

In this paper, we propose CLFuzz, a semantic-aware fuzzer for the implementation of vulnerability detection of cryptographic algorithms. Specifically, CLFuzz extracts the semantic information of cryptographic algorithms including their cryptographic-specific constraints and function signatures. Based on the above information, CLFuzz constructs high-quality test inputs that meet the complex input restrictions and can trigger error-prone boundary situations, significantly improving test effectiveness. Besides, CLFuzz applies logical cross-check across multiple test rounds based on the logical properties of algorithms that strengthens its logical bug detection ability and efficiency. Our evaluation results show that CLFuzz outperforms other state-of-the-art tools, and has detected many security-critical vulnerabilities in widely-used implementations of popular cryptographic algorithms.

## 9 ACKNOWLEDGEMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003), and Webank Scholar Project (20212001829).

## REFERENCES

[1] Pedram Amini and Aaron Portnoy. 2012. Sulley. https://github.com/OpenRCE/sulley.

[2] 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.

[3] ARM. 2021. Mbed TLS. https://tls.mbed.org/.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

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

[6] Paulo SLM Barreto, Hae Y Kim, Ben Lynn, and Michael Scott. 2002. Efficient algorithms for pairing-based cryptosystems. In *Annual international cryptology conference*. Springer, 354–369.

[7] John Richard Black Jr. 2000. *Message authentication codes*. University of California, Davis.

[8] blst. 2023. e1.c. https://github.com/supranational/blst/blob/master/src/e1.c#L270.
[9] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.
[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[11] Carmen Castro-Cabrera and Inmaculada Medina-Bulo. 2011. An approach to metamorphic testing for ws-bpel compositions. In *Proceedings of the International Conference on e-Business*. IEEE, 1–6.
[12] Lily Chen et al. 2008. Recommendation for key derivation using pseudorandom functions. *NIST special publication* 800 (2008), 108.
[13] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).
[14] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.
[15] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266
[16] CryptoPP. 2023. camellia.cpp. https://github.com/weidai11/cryptopp/blob/master/camellia.cpp.
[17] CryptoPP. 2023. hkdf.h. https://github.com/weidai11/cryptopp/blob/master/hkdf.h#L135.
[18] CVE-2021-23840. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-23840.
[19] Joan Daemen and Vincent Rijmen. 2002. *The design of Rijndael*. Vol. 2. Springer.
[20] Hans Delfs and Helmut Knebl. 2007. Symmetric-key encryption. In *Introduction to Cryptography*. Springer, 11–31.
[21] Yao Deng, Guannan Lou, Xi Zheng, Tianyi Zhang, Miryung Kim, Huai Liu, Chen Wang, and Tsong Yueh Chen. 2021. BMT: Behavior Driven Development-based Metamorphic Testing for Autonomous Driving Models. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*. 32–36. https://doi.org/10.1109/MET52542.2021.00012
[22] Morris Dworkin. 2001. *Recommendation for block cipher modes of operation. methods and techniques*. Technical Report. National Inst of Standards and Technology Gaithersburg MD Computer security Div.
[23] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. 1978. Message verification and transmission error detection by block chaining. US Patent 4,074,066.
[24] 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)*.
[25] FsCheck. 2022. FsCheck. https://github.com/fscheck/FsCheck.
[26] Google. 2015. American fuzzy lop. https://github.com/google/AFL.
[27] Google. 2019. Project Wycheproof. https://github.com/google/wycheproof.
[28] Google. 2021. ASAN. https://clang.llvm.org/docs/AddressSanitizer.html.
[29] Google. 2021. Sanitizers. https://github.com/google/sanitizers.
[30] Google. 2021. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[31] Ralph Guderlei and Johannes Mayer. 2007. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering* 17, 06 (2007), 757–781.
[32] guidovranken. 2021. Cryptofuzz. https://github.com/guidovranken/cryptofuzz.
[33] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.
[34] 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. https://doi.org/10.1145/3387940.3392252
[35] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. 2006. *Guide to elliptic curve cryptography*. Springer Science & Business Media.
[36] Christian Holler, Kim Herzig, Andreas Zeller, et al. 2012. Fuzzing with Code Fragments.. In *USENIX Security Symposium*. 445–458.
[37] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security* 1, 1 (2001), 36–63.
[38] Cameron F Kerry and Patrick D Gallagher. 2013. Digital signature standard (DSS). *FIPS PUB* (2013), 186–4.
[39] RSA Laboratories. 2021. PKCS #12: Personal Information Exchange Syntax Standard. https://web.archive.org/web/20140401120450/http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs12-personal-information-exchange-syntax-standard.htm.
[40] Xuejia Lai. 1992. *On the design and security of block ciphers*. Ph. D. Dissertation. ETH Zurich.
[41] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13.
[42] LLVM. 2022. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.
[43] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 46–56.
[44] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS protocol fuzzing: coverage guided packet crack and generation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[45] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[46] Microsoft. 2021. SymCrypt. https://github.com/microsoft/SymCrypt.

[47] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[48] Charlie Miller, Zachary NJ Peterson, et al. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* 4 (2007).

[49] MozillaSecurity. 2020. Peach. https://github.com/MozillaSecurity/peach.

[50] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. 2019. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 176–187.

[51] OpenSSL. 2021. OpenSSL. https://www.openssl.org/.

[52] OpenSSL. 2021. OpenSSL Demo. https://github.com/openssl/openssl/blob/master/demos/cipher/aesccm.c#L196.

[53] OpenSSL. 2023. tls1_prd.c. https://github.com/openssl/openssl/blob/master/providers/implementations/kdfs/tls1_prf.c#L186.

[54] OpenWall. 2022. yescrypt. https://www.openwall.com/yescrypt/.

[55] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[56] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. 39–50.

[57] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[58] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[59] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

[60] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*. Springer, 371–388.

[61] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.

[62] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 583–594.

[63] Tech-FAQ. 2022. Cryptographic Libraries. https://www.tech-faq.com/cryptographic-libraries.html.

[64] Henk CA Van Tilborg and Sushil Jajodia. 2014. *Encyclopedia of cryptography and security*. Springer Science & Business Media.

[65] 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.

[66] WolfSSL. 2022. WolfSSL. https://www.wolfssl.com/.

[67] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[68] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080

[69] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 769–786.