

ZTaint-Havoc: From Havoc Mode to Zero-Execution Fuzzing-Driven Taint Inference

YUCHONG XIE, Hong Kong University of Science and Technology, China

WENHUI ZHANG, Hunan University, China

DONGDONG SHE, Hong Kong University of Science and Technology, China

Fuzzing is a popular software testing technique for discovering vulnerabilities. A central problem in fuzzing is identifying hot bytes that can influence program behavior. Taint analysis can track the data flow of hot bytes in a white-box fashion, but it often suffers from stability issues and cannot run on large real-world programs. Fuzzing-Driven Taint Inference (FTI) is a simple black-box technique to track hot bytes for fuzzing. It monitors the dynamic program behaviors of program execution instances and further infers hot bytes in a black-box fashion. However, this method requires additional $O(N)$ program executions and incurs a large runtime overhead.

We observe that a widely used mutation scheme in fuzzing-havoc mode can be adapted into a lightweight FTI with *zero additional program execution*. In this work, we first present a computational model of the havoc mode that formally describes its mutation process. Based on this model, we show that the havoc mode can simultaneously launch FTI while generating and executing new testcases. Further, we propose a novel FTI called ZTaint-Havoc that doesn't need any additional program execution. ZTaint-Havoc incurs minimal instrumentation overhead of 3.84% on UniBench and 12.58% on FuzzBench, respectively. In the end, we give an effective mutation algorithm using the hot bytes identified by ZTaint-Havoc.

We conduct a comprehensive evaluation to investigate the computational model of havoc mode. Our evaluation result justifies that it is feasible to adapt the havoc mode to an efficient FTI without any additional program execution. We further implement our approach as a prototype ZTaint-Havoc based on the havoc mode of AFL++. We evaluate ZTaint-Havoc on two fuzzing datasets FuzzBench and UniBench. Our extensive evaluation results show that ZTaint-Havoc improves edge coverage by up to 33.71% on FuzzBench and 51.12% on UniBench over vanilla AFL++, with average improvements of 2.97% and 6.12% respectively, in 24-hour campaigns.

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

Additional Key Words and Phrases: Fuzzing, Fuzzing-Driven Taint Inference

ACM Reference Format:

Yuchong Xie, Wenhui Zhang, and Dongdong She. 2025. ZTaint-Havoc: From Havoc Mode to Zero-Execution Fuzzing-Driven Taint Inference. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA041 (July 2025), 23 pages. <https://doi.org/10.1145/3728916>

1 Introduction

Fuzzing has become a powerful technique for automated software testing. A core challenge in fuzzing is to identify program input bytes that can influence program behavior, commonly known as hot bytes. These hot bytes can determine variable values in the program's conditional branches.

Authors' Contact Information: Yuchong Xie, Hong Kong University of Science and Technology, Hong Kong, China, yxie@se.ust.hk; Wenhui Zhang, Hunan University, Changsha, China, zwenhui@hnu.edu.cn; Dongdong She, Hong Kong University of Science and Technology, Hong Kong, China, dongdong@cse.ust.hk.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2994-970X/2025/7-ARTISSTA041

<https://doi.org/10.1145/3728916>

When perturbing the value of `hot` bytes, a fuzzer is more likely to flip the program's branch conditions and further lead to new code coverage or new vulnerability discovery.

Traditional techniques to identify `hot` bytes for fuzzing tasks mainly rely upon dynamic taint analysis. Although it can effectively track `hot` bytes from program input, it suffers from large runtime overhead and fails to scale on large real-world programs[11, 38, 47]. To mitigate this issue, researchers propose a lightweight approach tailored for fuzzing, called fuzzing-driven taint inference (FTI)[6, 20, 32, 56]. FTI perturbs the program input, then monitors the difference in corresponding program behaviors when executing the program with the perturbed program input. Based on whether the discrepancy among the program behaviors exists after the program input perturbation, we can infer that the perturbed input bytes are `hot` bytes or not. In general, an effective FTI has two key requirements: (1) *Apply a small perturbation each time to ensure the newly generated program input is still valid*; (2) *Conduct a large number of perturbations such that the union of perturbed bytes covers all byte locations of program input*. Despite the fact that FTI can easily scale to large real-world programs, it incurs a large runtime overhead. Because they typically employ a byte-level taint inference scheme, i.e., perturb a single byte at a time, which requires *additional* $O(N)$ program executions for an input of size N . This substantial overhead raises a critical question: Is it possible to achieve effective taint inference with zero additional execution while maintaining the accuracy of existing approaches?

We observe that the havoc mode, a foundational mutation scheme that has been universally used in modern fuzzers that includes AFL[58], AFL++[18], libFuzzer[2], and libAFL[19], inherently satisfies the above two key requirements of FTI through its mutation characteristics. The havoc mode has two key properties: First, its mutation operators can apply minimal changes from the original seed, such as single-byte perturbations. Second, these operators select mutation positions uniformly across the byte locations of program input. These properties suggest the possibility of leveraging existing havoc mutations to build a novel FTI scheme, where every mutation in havoc mutations is considered as one inference instance in FTI.

To systematically understand and leverage these properties, we first present a computational model of the havoc mode, characterizing its mutation process through uniform random selection, stackable mutations, and splicing operations. Building upon this model, we propose a zero-execution FTI technique that integrates seamlessly with the havoc mode mutations. Our approach introduces two havoc-based techniques: (1) *an adaptive threshold k to adjust the degree of perturbation and normalize the byte influence distribution*, and (2) *a cumulative taint inference to prevent overtaint while achieving a uniform distribution of influenced bytes*.

To validate our approach, we implement a two-phase fuzzing algorithm ZTaint-Havoc. As shown in Figure 1, our method consists of two main phases: a sampling phase and a mutating phase. In the sampling phase, we apply our zero-execution FTI to identify `hot` bytes during normal havoc mode mutations. The mutating phase then leverages this information through a biased havoc mutation strategy focusing on these identified `hot` bytes. We evaluate our approach using two widely-adopted benchmarks: FuzzBench[36] and UniBench[31]. The results demonstrate that our method consistently outperforms AFL++, achieving up to 33.71% higher code coverage on FuzzBench and 51.12% on UniBench, with average improvements of 2.97% and 6.12%, respectively, in 24-hour fuzzing campaigns across diverse real-world applications. ZTaint-Havoc shows a runtime overhead of 3.84% on UniBench and 12.58% on FuzzBench, demonstrating that the additional computational cost remains within an acceptable range while achieving notable coverage gains.

This research makes the following contributions:

- We provide the first computational model of the havoc mode, revealing how its mutation characteristics naturally align with the FTI requirements.

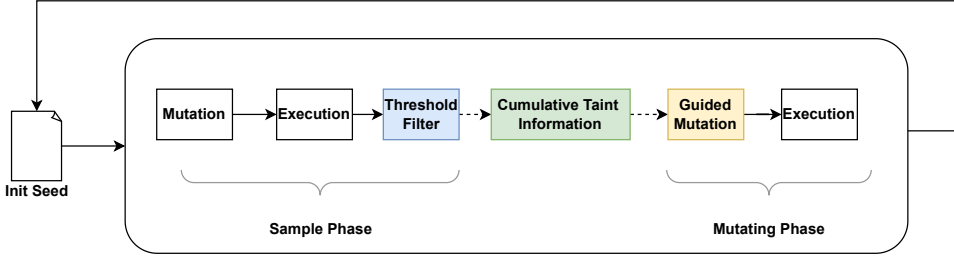


Fig. 1. Workflow of ZTaint-Havoc.

- We present a novel zero-execution FTI method that leverages havoc mode's inherent properties, eliminating the need for additional program execution.
- We demonstrate a practical implementation that enhances fuzzing effectiveness through taint-guided mutations, validated through a comprehensive evaluation. The code can be available at <https://github.com/Yu3H0/ZTaint-Havoc>

2 Background

2.1 Fuzzing-Driven Taint Inference (FTI)

Fuzzing-Driven Taint Inference (FTI) [6, 20, 32, 56] is a lightweight technique to identify hot bytes of program input tailored for fuzzing tasks. It first instruments the program at each branch statement to intercept the value of conditional variables that can influence the branch decision. Then it chooses a program input X with n bytes, represented as $X = \{x_1, x_2, \dots, x_n\}$. FTI records the value of conditional variables Y when dynamically executing the program with X . Iteratively, it modifies the i -th byte to obtain a mutant program input X_i , where $i = 1, 2, n$. In the end, it runs the program with X_i and compares the corresponding values of the conditional variables Y_i against Y . Once a discrepancy is observed, we can infer that the single-byte difference x_i between X_i and X can influence the program's branching behaviors. Hence, FTI considers the i -th byte x_i hot byte. FTI can effectively enhance the fuzzer's capability to solve complex branch constraints in the tested program. However, it usually incurs a large runtime overhead as a result of the additional $O(N)$ program executions. An effective FTI method needs to fulfill the following essential criteria:

R1: Minimal Difference Between Program Input and Mutant Input: The discrepancy between parent input and mutant input should be minimal. Large differences can result in divergence in program execution paths, hindering the value comparison of program's conditional variables. Ensuring the minimal difference is essential for precise FTI.

R2: Thorough Program Input Byte Examination: Every byte within the program input should be modified and investigated in a successful FTI. Omitting any byte locations could lead to under-tainting, where certain sections of the input are left unexamined. This thorough examination guarantees that no potential taint source is missed during FTI.

2.2 Havoc Mode

AFL's havoc mode, introduced by Zalewski[58], comprises a suite of operators for mutating input data and has been integrated into numerous adapted AFL fuzzers[9, 18, 35]. The havoc mode is invoked after a seed is selected for further mutation. In this process, a sequence of unit mutations is applied incrementally to the seed to generate a new testcase. Each unit mutation is performed by a randomly chosen havoc operator defined by the fuzzer, which typically incorporates many havoc operators such as flipping a bit, flipping a byte, increasing a byte value, or decreasing a byte value.

The sequence of unit mutations accumulates into a common havoc mode mutation that generates a new testcase. Specifically, the havoc mode first determines the total number of unit mutations that will be used to generate a new testcase, commonly denoted as the `havoc stack`. Then it randomly selects a sequence of havoc operators and incrementally applies them to the seed to generate the new test case. Furthermore, it implements a special mutation stage named the `splicing`, where the havoc mode mutation is not applied directly to an existing seed from the corpus, but to a newly generated seed combining two random seeds from the corpus.

3 White Box Interpretation

In this section, we provide a comprehensive white-box interpretation of havoc mode, diving into its key features and presenting our computational model. We begin by examining the three main components of havoc mode: `havoc stack`, `splicing` and `havoc operator`. Then, we analyze the core mechanism of the havoc mode, revealing how its stochastic nature and uniform distribution of mutations contribute to its magical power in exploring the input space. Finally, we give a formal description of the havoc mode's computation model.

3.1 Havoc Stack

The havoc stack denotes the total number of unit mutations to be applied incrementally to an initial seed when generating a new testcase. The default value of havoc stack varies on modern fuzzers due to different implementation details of havoc mode[2, 18, 58]. For instance, in AFL, the value is randomly selected from the set [2, 4, 8, 16, 32, 64, 128], whereas in AFL++ (v4.10c), the value is randomly selected from the range [1, 16] during the fuzzing process.

3.2 Splicing

Splicing is a common strategy in numerous fuzzers, appearing mainly as two forms: the splicing stage[58] and the splicing operator[2]. In fuzzers like AFL and AFL++, the splicing stage merges two seeds to produce a new initial seed for further mutation. This generally involves pinpointing the differences between two seeds and merging them from a random position within these differences, usually connecting the head of one seed with the tail of another. On the other hand, the splicing operator, used by fuzzers such as libFuzzer, serves as an independent mutation method. Both strategies strive to generate more varied inputs, potentially enabling fuzzers to explore novel regions of the input space that could be challenging to access through incremental mutations alone.

3.3 Havoc Operator

The havoc operator is a basic input mutation module in havoc mode. It takes a program input and outputs a mutant program input. Modern fuzzers implement many havoc operators, where each havoc operator applies a unique mutation scheme to the given input. A havoc operator typically selects a random position within the input byte sequence and then applies a specified mutation pattern on the selected byte locations. The following code snippet exemplifies a classic havoc operator, specifically the `MUT_FLIPBIT` operation:

```
u8 bit = rand_below(afl, 8);
u32 off = rand_below(afl, temp_len);
out_buf[off] ^= 1 << bit;
```

We observe that a havoc mode mutation consisting of a sequence of havoc operators is essentially a Markov process, where each havoc operator depends only on the output of the prior unit mutation. Formally, we define the havoc operator as a function $f(s)$ that takes input s and outputs m .

$$\text{Havoc Operator} : f(s) = m \quad (1)$$

We then define the perturbation distance $D(s, m)$ to measure the difference between two seeds s and m . The difference can be measured by Hamming distance[21], edit distance[28], or other sequence difference metrics. The perturbation distance between an initial seed s and its mutant m is approximately proportional to the havoc stack size h :

$$D(s, m) \approx h \cdot D(s, f(s)) \quad (2)$$

3.4 Havoc Mode's Computation Model

After explaining the three major components of the havoc mode in detail, we now give a formal description of the havoc mode's computation model as follows:

$$\text{Havoc Mode : } m = \begin{cases} f^h(s) = \underbrace{(f(f(\dots f(s) \dots)))}_{h \text{ times}} & \text{Normal Stage} \\ f^h(s') = \underbrace{(f(f(\dots f(s') \dots)))}_{h \text{ times}}, s' = \text{splice}(s) & \text{Splicing Stage} \end{cases} \quad (3)$$

The havoc mode consists of two stages: the normal stage that applies havoc mode mutation on a selected seed and the splicing stage that applies havoc mode mutation on a spliced seed. m represents the mutant testcase generated by the havoc mode mutation when given seed s . The function $f(s)$ denotes the havoc operator and $f^h(s)$ denotes the recursive iterations h times of the function $f(s)$ in seed s , representing the incremental perturbation of the havoc operators for h times in the common stage of the havoc mode. Moreover, the splicing operator $\text{splice}(s)$ is used during the splicing stage. It takes seed s as input and returns a spliced seed s' as output. There are two types of splicing operators. The first type is used during the splicing stage and is applied before other operators. The second type is implemented as a splicing operator in some fuzzers, such as AFL++ [18], and in such cases, we treat it as part of the havoc operator $f(s)$.

After havoc stack h is determined, we can represent the entire havoc mode as a Markov process. The state space is the set of all possible seeds. The transition matrix is the probability of the seed mutating to another seed. The initial state is the initial seed. The final state is the mutant. The goal of the Havoc mode is to find the mutant that can reach more basic block codes or trigger the target program to crash.

4 Connecting Havoc Mode with Fuzzing-Driven Taint Inference

In this section, we illustrate how the havoc mode theoretically satisfies the requirements of FTI. In the havoc mode, each seed goes through a sequence of havoc operators and generates many mutant testcases. Although not every mutant test case is necessarily suitable for FTI, there exist sufficient mutant testcases for an effective FTI. Specifically, we explain how the mutation characteristics of the havoc mode ensure the two critical requirements of FTI, as mentioned in Section 2.1.

Our insight is that the large number of mutant testcases generated by havoc mode can be used to perform an FTI without additional program execution. As shown in Figure 2, the havoc mode select and perturb almost every byte position of the seed after generating a large number of mutant testcases. Among these mutant testcases, many only have a single-byte difference from the seed. We observe that when the havoc stack is set to a small number like 1, the generated mutant testcase would only contain one-byte differences from the original seed. Since this single-byte mutation will eventually cover all byte positions due to the uniform selection strategy, these mutant testcases naturally satisfy both **R1** (minimal perturbation) and **R2** (comprehensive coverage) requirements.

Furthermore, while single-byte mutations enable an effective FTI, they may lead to undertaint issues in some corner cases. Consider a case where program behavior is triggered by multiple bytes simultaneously - for instance, when three input bytes undergo an AND operation. If the input

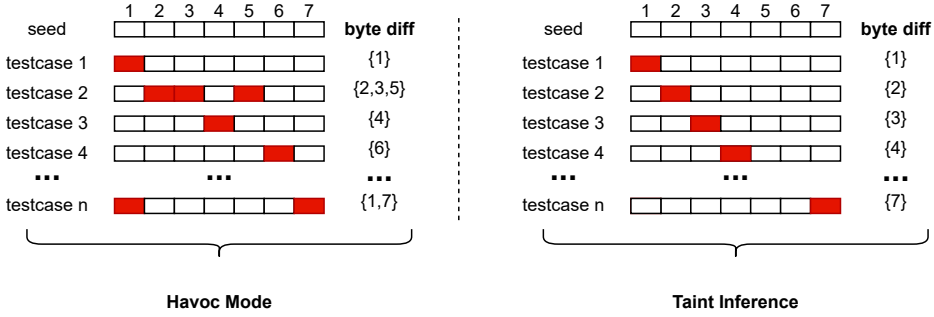


Fig. 2. An example illustrating how havoc mode can be used for FTI. The left side shows the havoc mutation process where testcases are generated with different bytes modified (marked in red) from a 7-byte seed, with each byte represented by a block. The right side demonstrates FTI process. Through multiple iterations, havoc mode covers all byte locations similar to FTI.

is '000', single-byte mutations can generate mutant testcases '100', '010' and '001' which might not influence the program state. Meanwhile, a simultaneous change to all three bytes '111' could significantly impact execution. In practice, the havoc stack in modern fuzzer like AFL++ is related to the mutation intensity, naturally producing larger perturbations when needed. This property ensures that our approach can capture multi-byte dependencies while still maintaining R1's similarity requirement through controlled havoc stack sizes.

5 Zero-Execution Fuzzing-Driven Taint Inference

We present our methodology for leveraging the havoc mode for zero-execution FTI:

- (1) **Mutation Threshold:** We introduce a parameter k to adjust the divergence between the mutant testcase and the initial seed. Specifically, this threshold limits the number of byte locations whose value differs between the mutant testcase and the input, thus maintaining a balance between exploration and inference accuracy.
- (2) **Cumulative Taint Inference:** We implement a mechanism to record the count of valid taint byte locations among the large number of mutant testcases. This approach allows for the progressive refinement of the taint data, leveraging the insights gained from each successive analysis to enhance the overall accuracy of the inference process.

In the following subsections, we discuss the mutation threshold and cumulative taint inference mechanisms in detail.

5.1 Mutation Threshold

To control the magnitude of divergence between the input seeds and their mutant test cases generated by havoc, we introduce an adaptive parameter k , which defines an upper bound to the number of byte differences. During normal stages, we directly use initial seeds as inputs. In contrast, in the splicing stage where two seeds are combined to create new test cases, we use the resulting spliced seeds as inputs to ensure stable starting points before havoc mutations. Formally, we define this constraint between seed *seed* and mutant testcases *mutant* as follows:

$$D_{\text{Hamming}}(\text{seed}, \text{mutant}) \leq k \quad (4)$$

where $D_{\text{Hamming}}(\text{seed}, \text{mutant})$ denotes the Hamming distance between two *seed* and *mutant*, representing the number of positions at which the corresponding bytes are different. The value of k is adaptively adjusted based on the seed length and mutation count in the havoc stage:

$$k = \beta \cdot \frac{L}{N_{\text{mut}}} \quad (5)$$

Here, L represents the byte length of *seed*, and N_{mut} is the number of mutant testcases to be generated by the havoc mode when given *seed*. The parameter β is a scaling factor that governs how k is computed based on these parameters.

This adaptive threshold mechanism offers several advantages:

- **Length-Aware Sampling:** By scaling k with the input length, we ensure havoc mode can cover almost all byte locations of the seed. A long seed typically is assigned a proportionally large k value, enabling more bytes to be modified within each individual mutant testcase.
- **Mutation-Aware Adjustment:** The adjustment based on mutation count ensures that when more mutations are planned, each mutation affects fewer bytes, leading to more controlled and evenly distributed changes across the seed.
- **Path Preservation:** By limiting the extent of mutations, we mitigate the risk of generating mutants that deviate excessively from the initial seed. This helps preserve the validity of execution-path comparisons, which is crucial for accurate taint inference.
- **Distribution Normalization:** While havoc operations are theoretically designed to modify bytes uniformly, certain fuzzers employ operators like `MUT_OVERWRITE_COPY` that can skew this distribution. Such operators first select a variable length and then choose a mutation position, resulting in a non-uniform distribution of influenced bytes, typically with higher frequency in the middle and lower at the extremities. By imposing a threshold k , we effectively normalize this distribution.

5.2 Cumulative Taint Inference

Building upon the mutation threshold mentioned above, we propose a cumulative taint inference method to further improve accuracy and mitigate overtaint issues. We denote the set of bytes in the input that truly influence program execution (true taints) as T , the set of input bytes that do not affect program execution (irrelevant inputs) as I , and the set of bytes that differ in each inference process as D .

Note that we cannot treat the inferred taint bytes from each iteration as equivalent, nor can we use a single aggregate set to represent tainted bytes. This is because the uniform distribution of havoc operators would eventually cover the entire input space, even if only elements of T affect program execution, leading to over-tainting.

To address this challenge, we propose a data structure: an array of the same length as the input, with each element initially set to zero. During each inference iteration, we increment the value at the index corresponding to each byte in D . The advantage of this approach lies in its ability to differentiate between true taints and noise. Since each inference result invariably includes elements from T , after multiple iterations, the values corresponding to elements in T will be consistently higher than those corresponding to elements in I . This method allows us to obtain a more accurate approximation of T , thereby reducing the impact of over-tainting.

6 Enhanced Havoc Mode with Zero-Execution FTI

Our approach bifurcates the havoc process for each input into two distinct phases in 1:

- (1) Fuzzing-driven taint inference, as described in the preceding sections.
- (2) Taint-guided mutation, which leverages inferred taint information.

Algorithm 1 Enhanced Havoc Mode with Zero-Execution FTI

```

1: Initialize  $taint\_count[1..n] \leftarrow 0$  where  $n$  is input length
2:  $seed \rightarrow$  initial seed
3: /* Phase 1: Vanilla havoc mode with zero-execution FTI */
4:  $pb_{seed} = execute(seed)$  ▷ Record program behavior for  $seed$ 
5: for  $i = 1, 2, \dots, n_1$  do
6:    $t_i = havoc\_mutate(seed)$  ▷ Vanilla havoc mode generates a testcase  $t_i$ 
7:    $pb_i = execute(t_i)$  ▷ Record program behavior for  $t_i$ 
8:    $ByteDiff = diff(t_i, seed)$ 
9:   if  $|ByteDiff| \leq k$  then ▷ Skip testcases with too many byte diffs to avoid overtaint
10:    if  $pb_i \neq pb_{seed}$  then
11:      for each index  $j$  in  $ByteDiff$  do
12:         $taint\_count[j] \leftarrow taint\_count[j] + 1$ 
13: /* Phase 2: Biased havoc mode focusing on bytes with high taint count */
14: for  $i = 1, 2, \dots, n_2$  do
15:    $t_i = biased\_havoc\_mutate(seed, taint\_count)$  ▷ Mutate bytes based on taint count
16:    $execute(t_i)$ 

```

Upon completion of the first phase, we obtain a data structure $taint_count$, which encapsulates the cumulative count of inferences for each byte position. This structure serves as the foundation for our taint-guided mutation strategy.

We propose a novel method that adapts the existing havoc operators to incorporate taint information. The core of this adaptation lies in the modification of the random selection process used by these operators. Specifically, we replace the uniform random function $rand()$ with a variant aware of the taint that aligns with the distribution implied by $taint_count$. Formally, we define our taint-guided random function as follows:

$$P(i) = \frac{taint_count[i]}{\sum_{j=1}^n taint_count[j]} \quad (6)$$

where $P(i)$ represents the probability of selecting the i -th byte as the starting point for a mutation operator, and n is the length of the input.

This probabilistic selection mechanism ensures that bytes with higher taint counts, those more likely to influence program behavior, have a correspondingly higher probability of being chosen as the focal points for mutation operations. Consequently, our mutation strategy becomes more targeted, concentrating on the most influential parts of the input as identified by our taint inference process.

7 Implementation

In this section, we introduce ZTaint-Havoc, our proof-of-concept implementation designed for zero-execution FTI. We use the AFL++ version 4.10c, which is the latest and most performant version of the widely recognized state-of-the-art fuzzer[1, 3, 7], as the base fuzzer.

ZTaint-Havoc enhances the AFL++ LLVM-based instrumentation pass to extract the intraprocedural control flow graph (CFG) for each function and incorporates its metadata into the binary. During a fuzzing campaign, we dynamically label nodes in the CFG as visited or unvisited based on coverage data. We also maintain an updated list of frontier nodes, concentrating exclusively on frontier branches, a subset of frontier nodes that includes control instructions leading to conditional jumps. For each control instruction, we instrument an instruction or a function to detect changes in the control condition. For string comparison types, such as `strcmp` or `memcmp`, we insert a function, while for `icmp` types, we insert a single instruction. To reduce run-time overhead, we add an adaptive switch for each hook, only enabling the hook for frontier branches.

In the mutation part, when a seed is selected as an initial seed for havoc, we first run the seed in the target program and record the frontier branch and the corresponding control instruction information. Then we calculate an adaptive threshold k bounded between 1 and $\lfloor \text{len}(s)/32 \rfloor$, where $\text{len}(s)$ is the length of the initial seed. During the havoc stage, after the first run, the first 50% mutations are seen as sampling, and we only concentrate on mutations that differ within k bytes compared with the initial seed. After these mutations are executed, we record the frontier branch and the corresponding control instruction information again. If the control instruction information is different from the initial seed, we will taint the 5 different bytes. Specifically, we will maintain a taint array; each time we taint a byte, we will increment the corresponding position in the taint array by 1. After the sampling stage, we will use the taint information to guide the mutation. We will utilize the taint array to assess the probability of mutation for each byte. A higher value indicates an increased likelihood that that byte is selected as the starting point for mutation.

8 Evaluation

In this section, we aim to answer the following research questions:

- **RQ1:** Does havoc mode follow a uniform distribution, and how does the introduction of thresholds affect this distribution?
- **RQ2:** How does uniform distribution compare to alternative distributions in terms of effectiveness?
- **RQ3:** How does the havoc stack size impact the perturbation distance between the seed input and its mutants?
- **RQ4:** What is the impact of splicing operations on perturbation distance?
- **RQ5:** How does ZTaint-Havoc perform in terms of code coverage compared to AFL++?
- **RQ6:** How does the threshold parameter k affect the fuzzing performance in code coverage?
- **RQ7:** What is the runtime overhead of ZTaint-Havoc compared to the baseline?
- **RQ8:** How does ZTaint-Havoc compare to CMPLOG in terms of code coverage?

8.1 Benchmark

For our experiment, we chose FuzzBench[36] and UniBench dataset[31]. FuzzBench has established itself as a standard benchmark in fuzzing research[7, 10, 33], providing a controlled environment with well-designed harnesses and standardized evaluation metrics. To further enhance the comprehensiveness of our evaluation, we also incorporated standalone programs from the UniBench dataset, which features a diverse collection of real-world applications processing various data formats including images, videos, audio files, text documents, and binary files. We selected 19 targets from the FuzzBench dataset and 17 targets from the UniBench dataset, as the remaining programs have compilation issues, such as dependency incompatibility. The target list is shown in Table 1.

8.2 Mutation Distance

In this section, we assess the perturbation level using four distinct distance metrics: the L0-norm, L1-norm, L2-norm, and the edit distance (Levenshtein distance). Each of these metrics provides unique insights into the nature of mutations introduced by fuzzing techniques.

Let x denote our initial seed, and y represent the post-disturbance corpus. When the lengths of x and y differ, we pad the shorter sequence with zeros to ensure equal lengths. We then transform them into two arrays, X and Y , where each element ranges from 0 to 255. The length of these arrays is denoted as n , which is the length of the sequences after padding. The difference array $Diff$ is computed as:

$$Diff_i = X[i] - Y[i] \quad \text{for } i = 1, 2, \dots, n \quad (7)$$

Table 1. Studied programs in our evaluation.

Targets	Version	# Edge	Targets	Version	# Edge	Targets	Version	# Edge
FuzzBench Targets			sql	c78cbf2	77,147	tcpdump	4.8.1	32,760
bloaty	52948c1	163,038	systemd	07faa49	1,059	jq	jq-1.5	6,177
zlib	zlib-1.2.13	4,634	jsoncpp	8190e06	9,820	sqlite3	SQLite-3.8.9	67,533
lcms	f0d9632	13,649	libjpeg	b0971e4	17,783	cflow	cflow-1.6	7,690
libpcap	c639612	15,450	libpng	cd0ea2a	9,032	exiv2	exiv2-0.28.0	122,536
libxml2	c7260a4	82,498	libxslt	180cdb8	60,091	ffmpeg	ffmpeg-4.0.1	586,737
openh264	fa6d099	18,677	openssl	b0593c0	85,304	infotocap	ncurses-6.1	11,470
re2	4be2407	15,173	UniBench Targets			nm-new	binutils-2.28	57,709
stbi	5736b15	7,655	mp42aac	1.5.1-628	21,707	objdump	binutils-2.28	77,318
vorbis	84c0236	12,895	lame	lame-3.99.5	15,776	pdftotext	xpdf-4.00	49,169
woff2	9476664	19,609	mujs	mujs-1.0.2	21,288	mp3gain	1.5.2	3,344
curl	a20f74a	122,327	flvmeta	flvmeta-1.2.1	6,512	tiffsplit	tiff-3.9.7	12,527
harfbuzz	cb47dca	78,196				jhead	3.00	2,092

For the L0, L1, and L2 norms, we employ the generalized Lp-norm [49]:

$$\|\text{Diff}\|_p = \left(\sum_{i=1}^n |\text{Diff}_i|^p \right)^{\frac{1}{p}} \quad (8)$$

where L0 counts non-zero elements, representing the number of changed bytes (note that L0 is not a true norm as it violates norm properties); L1-norm ($p = 1$) sums absolute differences, measuring total change magnitude; and L2-norm ($p = 2$) is the Euclidean distance, giving more weight to larger changes.

Additionally, we introduce the edit distance, defined as the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into another. It is particularly useful for comparing strings of different lengths and capturing structural changes. By employing these complementary metrics, we can comprehensively analyze the nature and extent of mutations introduced by various fuzzing techniques, providing a multi-faceted view of their effectiveness and characteristics.

8.3 RQ1: Uniform Sampling

8.3.1 Experimental Setup. To validate our hypotheses regarding mutation distributions and the effectiveness of our proposed threshold mechanism, we conducted a series of experiments that focused on two key dimensions: the distribution of mutation **starting positions** and the distribution of **influenced bytes**. The havoc mode is essentially a string edit process using a set of fixed rules, as formally described in Section 3.4. It generates many mutant inputs when given a seed input. Note that this mutant generation step is orthogonal to program logic because the variance in program logic does not influence how mutant inputs are generated. Specifically, the program logic is only used to determine the code coverage-based feedback after the mutant generation. Our evaluation collects data from the mutant generation phase. We selected 10 representative programs with varying seed lengths to maintain experimental feasibility. For each target, we selected an initial seed of moderate length from those provided by FuzzBench and UniBench. Table 2 shows the lengths of the selected seeds. Our experiments involved executing 100,000 iterations of the havoc stage using a single seed for each target. For each target program, we conducted 10 independent measurements to ensure statistical reliability.

Table 2. Seed length of the selected programs. * indicates targets from UniBench.

Binary	Length	Binary	Length
cflow*	38,139	vorbis	2,603
bloaty	22,120	openssl	1,114
pdftotext*	12,567	tcpdump*	531
tiffsplit*	9,834	libpcap	220
ffmpeg*	6,714	lcms	128

Our first experiment aimed to verify the uniformity of mutation starting positions. We maintained the stack size settings of vanilla AFL++ and focused on the 28 substitution operators among AFL++’s 37 havoc operators, as these preserve seed length. By tracking the starting position of each mutation operation, we sought to demonstrate that the selection of starting positions is approximately uniform across the input space.

The second dimension of our analysis focused on the distribution of influenced bytes resulting from mutation operations, where influenced bytes are defined as bytes that differ between the original seed and the mutated input. We only consider mutants that maintain identical execution paths as their initial seeds. We conducted two sets of experiments: one without constraints and the other applying a threshold of $k = 5$. These experiments aimed to demonstrate that without a threshold, the distribution of influenced bytes is nonuniform, whereas imposing a threshold leads to a more uniform distribution across the input space.

8.3.2 Observations. Figure 3 presents the results of our first experiment. The data demonstrate a predominantly uniform distribution of mutation start positions across the majority of the input space. However, a notable decrease in mutation frequency is observed near the tail of the input. This tail-end decline can be attributed to constraints imposed by certain mutation operators, such as MUT_OVERWRITE_FIXED, which have limited selection options near the input’s end due to length constraints.

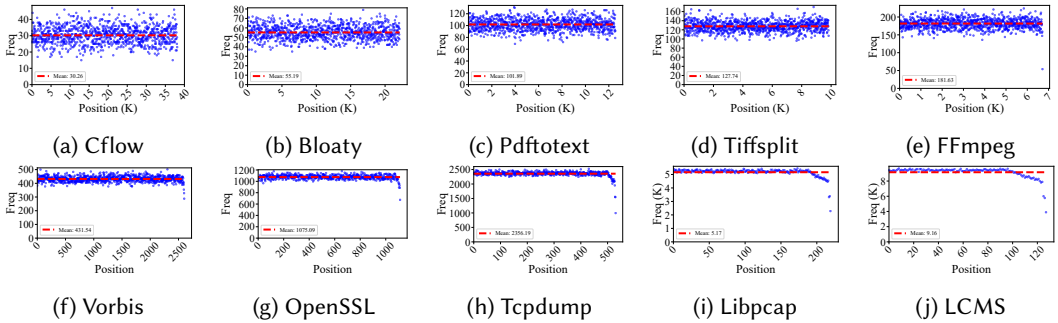


Fig. 3. Distribution of mutation starting positions across ten target programs. The x-axis represents the position in input space where mutations begin, while the y-axis shows the accumulated selection frequency over 10 independent runs. The red line indicates the average frequency across all positions.

Figure 4 presents the results of the first part of our second experiment, which examined the distribution of influenced bytes during the AFL++ havoc stage without applying any threshold. For most targets, the distribution exhibits a pronounced peak, resembling a mountain-like shape rather than a uniform distribution. This pattern clearly demonstrates that in practice, influenced bytes

are not uniformly distributed across the input space during havoc mutations. Some regions of the input are significantly more likely to be affected by mutations than others. The irregular shape of these distributions varies across different seeds, as it is highly dependent on the specific content of each seed input.

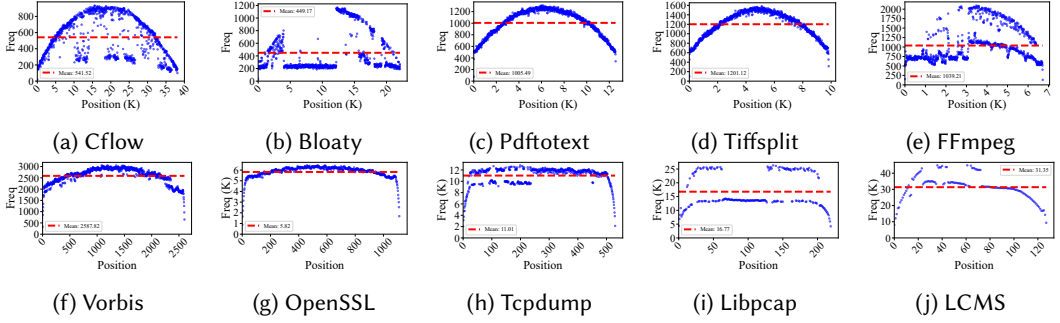


Fig. 4. Distribution of influenced bytes (bytes that differ between the original seed and mutated input) across ten target programs. The x-axis represents the byte position in input space, while the y-axis shows the accumulated frequency of bytes being influenced over 10 independent runs, demonstrating the distribution pattern without threshold constraints. The red line indicates the average frequency across all positions.

The second part of our experiment investigated the distribution of influenced bytes after implementing a threshold mechanism. Figure 5 illustrates these results. In stark contrast to the previous findings, the distribution with the threshold applied exhibits a markedly more uniform pattern across most of the input space. The pronounced peaks observed in the non-threshold scenario have been largely eliminated, with only narrow regions at the beginning and end of the input showing slight deviations from uniformity. This nearly uniform distribution for most of the input demonstrates the effectiveness of our threshold approach in equalizing the influence of mutations across the input space. The results clearly indicate that our threshold mechanism successfully mitigates the bias towards specific input regions observed in standard havoc mutations.

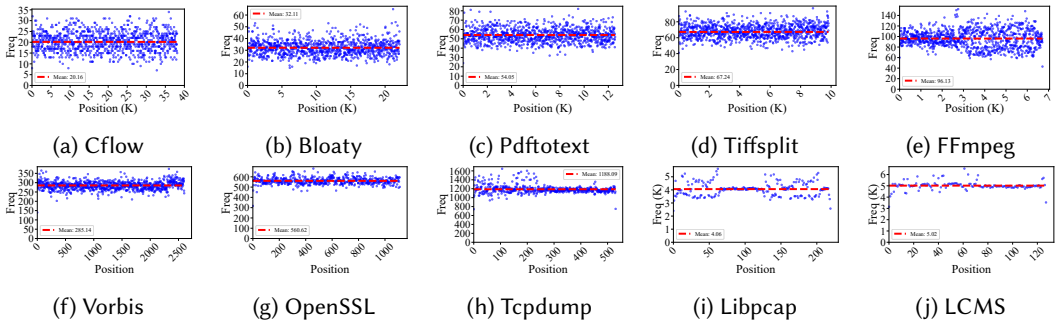


Fig. 5. Distribution of influenced bytes across ten target programs with a perturbation threshold of $k=5$. The x-axis represents the byte position in input space, while the y-axis shows the accumulated frequency of bytes being influenced over 10 independent runs, demonstrating the distribution pattern under controlled mutation impact. The red line indicates the average frequency across all positions.

Table 3. **Coverage comparison between vanilla AFL++ and AFL++ with normally distributed starting positions (Normal) across FuzzBench and UniBench targets. Each number represents the average edge coverage from 10 one-hour independent runs, with bold numbers indicating higher coverage.**

Targets	Vanilla	Normal	Targets	Vanilla	Normal	Targets	Vanilla	Normal
FuzzBench Targets			sql	9887	9817	tcpdump	924	893
bloaty	6261	6108	systemd	2122	2106	jq	1689	1666
zlib	699	690	jsoncpp	906	870	sqlite3	27416	27253
lcms	461	195	libjpeg-t	2749	2722	cflow	2018	2011
libpcap	42	36	libpng	1549	1440	exiv2	1947	1894
libxml2	4595	4283	libxslt	2761	2686	ffmpeg	20621	20509
openh264	6209	5959	openssl	11034	11031	infotocap	1105	1093
re2	4044	3896	UniBench Targets			nm-new	1393	1351
stbi	1239	1016				objdump	4411	4331
vorbis	1517	1478	mp42aac	1362	1320	pdftotext	5417	5399
woff2	1794	1752	lame	5379	5184	mp3gain	1260	1250
curl	8186	7790	muj	4145	4018	tiffsplit	1207	1175
harfbuzz	24300	23998	flvmeta	375	364	jhead	334	327

Result 1: Both the havoc's starting positions and the influenced bytes (with threshold) exhibit approximately uniform distributions across the input space.

8.4 RQ2: Effect of Uniform Sampling

8.4.1 Experimental Setup. For this experiment, because we use coverage as the metric, we choose all targets in FuzzBench and UniBench datasets to obtain a comprehensive evaluation. For our evaluation, we selected moderate-length seeds and well-formed structures from the seed corpora of AFL++, FuzzBench, and UniBench. We compared two configurations: vanilla AFL++ and AFL++ with normally distributed starting positions centered at randomly chosen positions in the seed (with the center fixed at fuzzer initialization). Each target was evaluated for 1 hour with 10 independent runs to account for the inherent randomness in fuzzing. To eliminate the impact of any overhead introduced by our modifications, we compare the coverage with the same number of executions by using the smaller execution count between the two configurations in the 1-hour runs.

8.4.2 Observations. Based on our experimental results across all tested targets from FuzzBench and UniBench datasets, we observe that vanilla AFL++ (using uniform distribution for selecting mutation positions) consistently achieves higher coverage compared to using normally distributed starting positions, with an average improvement of 4.69%. This suggests that the uniform selection of mutation positions provides better exploration of the input space and more adaptiveness across different program structures, rather than concentrating mutations around fixed points.

Result 2: Uniformly choosing mutation starting position contributes to the effectiveness and adaptiveness for the havoc mode.

8.5 RQ3: Effect of Mutation Stack

8.5.1 Experimental Setup. In order to exclude other influences, we use the same targets, initial seed, and number of executions used in Section 8.3. We use five sets of experimental setups, that is, the havoc stack is 1, 2, 4, 8, and 16 for five cases. For each target program, we conducted 10 independent measurements to ensure statistical reliability. We measure the average perturbation distance using four different metrics: L0-norm, L1-norm, L2-norm, and edit distance, calculated

between the initial seed and the mutant. All experiments are conducted using AFL++ as our base fuzzer.

8.5.2 Observation. The results are presented in Figure 6, which includes separate graphs for each metric. In these ten targets, the perturbation distance exhibits an increase with the increase in the size of the havoc stack. The results indicate a clear correlation, demonstrating that the havoc stack effectively amplifies the perturbation distance.

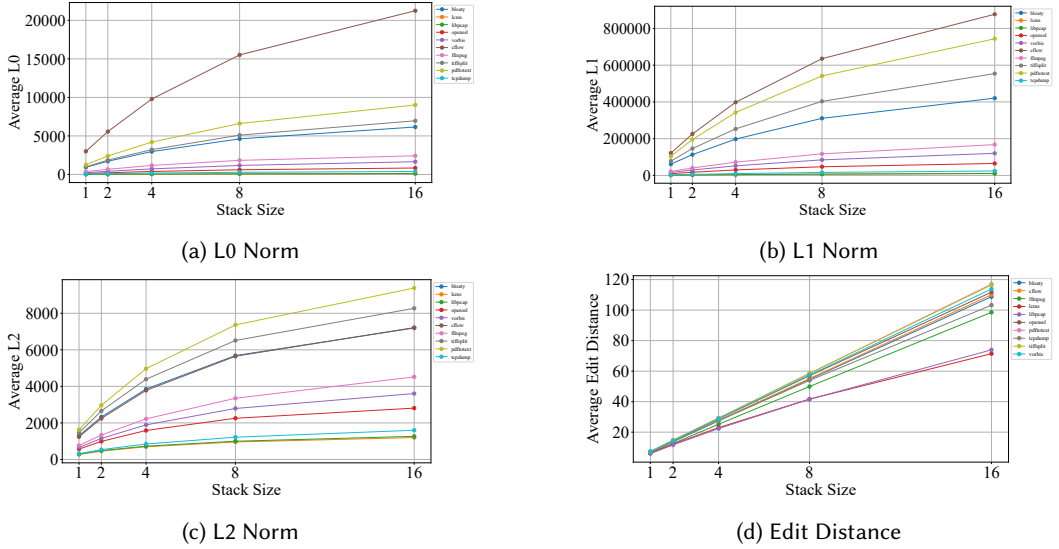


Fig. 6. **Changes in perturbation distances across different havoc stack sizes (1, 2, 4, 8, 16).** Each line represents a target program's perturbation distance measured by L0, L1, L2 norms and edit distance. Results from all ten targets are plotted individually.

Result 3: The perturbation distance increases with larger havoc stack size.

8.6 RQ4: Effect of Splicing

8.6.1 Experimental Setup. We designed our experiments to thoroughly investigate the effects of the splicing operator and the splicing stage on perturbation distance. We chose AFL++ as our measurement fuzzer due to its implementation of both the splicing operator and the splicing stage. To maintain consistency and minimize external variances, we used the same targets and initial seeds as in our previous experiments. For each target program, we conducted 10 independent measurements to ensure statistical reliability.

For testing splicing stage, we set the havoc stack to vanilla settings. The experiment consisted of 20,000 executions: the first 10,000 in normal stage to generate a diverse seed pool, followed by 10,000 in splicing stage. This design ensures the availability of two seeds for the splicing operation. We then measured the average perturbation distance of the resulting mutants. To evaluate the splicing operator, we set the havoc stack to 1 and compared the average perturbation distance produced by the splicing operator against other operators. In both experiments, we maintained the same measurement dimensions as in our previous studies, utilizing four metrics to quantify perturbation distances: L0-norm, L1-norm, L2-norm, and edit distance.

8.6.2 Observation. Splicing Stage The results are presented in Table 4. Using all four metrics, our analysis reveals that across all 10 target programs, the perturbation distances in the splicing stage consistently exceed those in normal stage. This observation confirms that the splicing stage introduces larger mutations compared to the normal stage.

Splicing Operator The results are presented in Table 5. The data show that the splicing operator generates consistently larger perturbation distances compared to other operators across all targets.

Result 4: Both splicing stage and splicing operator generate larger perturbation distances compared to normal stage and other operators, respectively.

Table 4. **L0-norm, L1-norm, L2-norm and edit distance in normal stage (-N suffix) and splicing stage (-S suffix). Bold numbers indicate the higher value between normal and splicing stages.**

Targets	L0-N	L0-S	L1-N	L1-S	L2-N	L2-S	Edit-N	Edit-S
cflow	10,238	15,041	418,382	673,027	3,802	6,168	36	8,898
bloaty	2,864	10,627	191,824	1,020,463	3,626	11,615	33	7,581
pdfotext	4,088	5,981	334,316	500,484	4,697	7,556	34	4,287
tiffsplit	3,745	5,606	295,852	476,705	4,852	7,468	44	3,013
ffmpeg	897	3,124	55,101	461,475	1,719	9,800	19	2,159
vorbis	736	1,587	53,111	139,519	1,841	4,081	34	1,194
openssl	402	741	30,973	60,058	1,562	2,714	35	489
tcpdump	149	304	8,345	23,111	699	1,594	26	232
libpcap	52	81	4,238	7,291	688	1,067	25	66
lcms	40	75	4,221	7,291	633	949	24	70

Table 5. **L0-norm, L1-norm, L2-norm and edit distance in other mutation operators (-O suffix) and splicing operator (-S suffix). Bold numbers indicate the higher value between other operators and splicing operator.**

Targets	L0-O	L0-S	L1-O	L1-S	L2-O	L2-S	Edit-O	Edit-S
cflow	2,308	7,565	93,400	306,402	971	3,047	6	15
bloaty	721	2,307	47,398	152,006	1,025	3,113	6	16
pdfotext	1,014	3,239	82,464	263,318	1,306	4,038	6	16
tiffsplit	780	2,472	61,489	195,378	1,169	3,586	6	17
ffmpeg	286	898	16,942	53,505	619	1,721	5	14
vorbis	172	555	12,380	39,841	524	1,555	6	16
openssl	93	301	7,124	23,232	453	1,335	6	16
tcpdump	44	140	2,318	7,300	264	685	6	16
libpcap	13	38	1,031	2,867	259	551	5	13
lcms	9	30	992	3,143	227	575	5	14

8.7 RQ5: Zero-Execution Fuzzing-Driven Taint Inference

8.7.1 Experimental Setup. To achieve a comprehensive evaluation, we leverage all targets from the FuzzBench and UniBench datasets. Since GreyOne[20] and ProFuzz[56] are not open-source, we compare our approach with state-of-the-art fuzzers[3] to demonstrate the effectiveness of our method. We assess the coverage of AFL++ and ZTaint-Havoc under two different configurations: 1) default settings with no additional features, and 2) enhanced settings with both dictionary and AFL++ cmplog mode[6] enabled. For each target, following previous research and typical fuzzing evaluation methodologies[27, 34, 59, 62], we conduct 10 campaigns of 24 hours for each pair of

fuzzer programs. To maintain fairness in our comparison, each fuzzer operates on a single core per fuzzing campaign. We use the same seeds as Section 8.4. Our analysis includes standard statistical summaries, such as the mean over our evaluation metrics. Furthermore, we employ the Mann-Whitney U test to verify that the observed performance differences are statistically significant. This nonparametric test does not rely on assumptions about distribution, making it a popular choice in software testing studies to evaluate randomized algorithms[5] and fuzzers[6].

8.7.2 Observation. FuzzBench Targets Across FuzzBench targets (see results in Table 6), ZTaint-Havoc demonstrates superior performance compared to AFL++ in 13 out of 19 programs, achieving mean coverage improvements. For instance, `lcms` shows a remarkable 33.71% coverage increase over AFL++. When comparing ZTaint-Havoc + Cmplog + Dict to AFL++ + Cmplog+Dict, the performance gap remains consistent, with ZTaint-Havoc + Cmplog + Dict outperforming AFL++ + Cmplog+Dict on 13 out of 19 programs. Notable improvements include a 8.63% increase for `libxslt`. The mean gain of ZTaint-Havoc over AFL++ is 2.97%, while ZTaint-Havoc + Cmplog + Dict achieves a 1.65% mean gain over AFL++ + Cmplog+Dict. As noted in previous research[41], the performance gap on FuzzBench is typically smaller than on standalone programs due to FuzzBench’s well-optimized baseline implementations. However, in some cases such as `stbi`, AFL++ slightly outperforms ZTaint-Havoc. These exceptions may be attributed to challenges with nested conditions, instrumentation overhead that affects throughput, and the lack of specialized constraint solvers for taint-guided mutation.

Standalone Targets Comparison of mean coverage achieved by ZTaint-Havoc against AFL++, ZTaint-Havoc + Cmplog + Dict against AFL++ + Cmplog+Dict is presented in Table 6. Across standalone programs, ZTaint-Havoc exhibits improvements over AFL++ on 11 programs, achieving up to 6.12% more code coverage on average across the standalone target set. One of the targets where we see notable improvement is `nm-new`, where ZTaint-Havoc uncovers 51.12% more edges than AFL++. Similarly, ZTaint-Havoc + Cmplog + Dict outperforms AFL++ + Cmplog+Dict on 10 programs, including a remarkable 41.53% improvement on `mp42aac`. But in some cases such as `infotocap`, AFL++ slightly outperforms ZTaint-Havoc. These exceptions may be attributed to the same reason as discussed before.

Result 5: ZTaint-Havoc outperforms existing state-of-the-art fuzzers with average improvements of 6.12% on UniBench and 2.97% on FuzzBench targets.

8.8 RQ6: Threshold parameter k

8.8.1 Experimental Setup. To achieve a comprehensive evaluation, we leverage all targets from the FuzzBench and UniBench datasets. To investigate the impact of different k values on ZTaint-Havoc’s performance, we configure six different settings: five fixed k values ($k=1$, $k=10$, $k=100$, $k=1000$, and $k=10000$) and one adaptive k setting. For each target program, we conduct concurrent runs across all settings using the same seed inputs as in previous sections. Each configuration is tested with 10 independent instances running for 8 hours.

8.8.2 Observation. As shown in Table 7, the adaptive k setting achieves the highest global wins (`glo_win`) across all targets, where `glo_win` represents the number of targets for which a particular setting achieves the highest average coverage. For constant k values, there is no clear trend in `glo_win` as k increases or decreases: $k=1$, $k=10$, $k=100$, $k=1000$, and $k=10000$ achieve `glo_win` values of 5, 5, 2, 9, and 7 respectively. This observation suggests that no single fixed k value consistently outperforms others across different targets. The results indicate that adaptive k effectively adjusts to the specific characteristics of each target program, eliminating the need for manual parameter tuning.

Table 6. Mean edge coverage of ZTaint-Havoc and AFL++, as well as ZTaint-Havoc + Cmplog + Dict (ZH+CD) and AFL++ + Cmplog+Dict (AFL+CD) on 19 FuzzBench programs (left) and 17 standalone programs (right) for 24 hours over 10 runs. We mark the highest number in bold for each comparison.* indicates statistically significant differences.

Targets	ZTaint-Havoc	AFL++	ZH+CD	AFL+CD
bloaty	3,165	3,105	2,763	2,736
curl	14,089	14,049	16,158	16,408
libxslt	4,755	4,682	5,613*	5,167
harfbuzz	23,696	23,432	25,039	25,627
jsoncpp	1,338	1,338	1,343	1,343
lcms	1,652	1,235	2,713	2,547
libjpeg	3,244	3,177	3,240	3,235
libpcap	45	50	4,438	4,307
libpng	2,306	2,281	2,656	2,670
libxml2	10,896	9,128	13,332	12,764
vorbis	2,044	2,045	2,039	2,033
openh264	13,543	13,567	11,560	10,716
openssl	4,651	4,625	4,628	4,634
woff2	2,395	2,356	2,550*	2,489
zlib	881	884	905	889
re2	6,260	6,253	6,261*	6,199
sqlite3	14,087	13,880	16,670	16,617
stbi	2,112	2,132	2,842	2,911
systemd	2,533	2,525	2,490	2,474
Mean gain	2.97%		1.65%	

Targets	ZTaint-Havoc	AFL++	ZH+CD	AFL+CD
exiv2	5,688	4,776	7,130	6,623
tiffsplit	2,503*	2,173	2,338*	2,156
mp3gain	1,554	1,552	1,578	1,574
mujs	6,616	6,696	6,921	6,838
pdftotext	6,838	6,771	6,936	7,063
infotocap	2,139	2,317	2,263	2,448*
mp4aac	1,843*	1,739	2,903*	2,051
flvmeta	477	477	476	476
objdump	5,578*	5,429	4,973	4,973
tcpdump	8,078*	7,146	6,362	5,851
ffmpeg	22,221	21,734	18,986	18,502
jq	3,529	3,548	3,368	3,383
cflow	2,118	2,122	2,078	2,080
nm-new	3,386*	2,240	1,431	1,441
sqlite3	15,682	15,220	14,637	14,273
lame	5,732	5,731	5,559	5,467
jhead	361	361	378	372
Mean gain	6.12%		3.86%	

We attribute these observations to two main factors. First, as k increases, the overhead for tracking byte-level differences becomes more significant, leading to reduced overall throughput in the fuzzing process. Second, the choice of k value presents a fundamental trade-off: a small k may result in undertaint issues where potential byte dependencies are missed, while a large k can lead to overtaint problems where excessive bytes are incorrectly identified as related, causing inefficient mutation attempts on irrelevant bytes.

Result 6: The performance of ZTaint-Havoc varies with different k values, with the adaptive k setting demonstrating better applicability across diverse targets compared to fixed k values.

8.9 RQ7: Evaluation of ZTaint-Havoc's Runtime Overhead

8.9.1 Experimental Setup. To evaluate the runtime overhead introduced by ZTaint-Havoc's fuzzing-driven taint inference mechanism, we directly utilize the execution count data from our experiments in Section 8.7. Specifically, we compare the number of executions achieved by ZTaint-Havoc and AFL++ over 10 independent 24-hour runs across both FuzzBench and UniBench targets. This metric serves as a direct indicator of the runtime overhead, as any additional computational cost from fuzzing-driven taint inference would manifest as reduced execution throughput. The experimental configuration, including the hardware setup, seed inputs, and single-core execution environment, remains identical to that described in Section 8.7.

8.9.2 Observation. As shown in Table 8, the overhead varies between the two benchmark suites: UniBench shows a lower overhead of 3.84%, while FuzzBench exhibits a higher overhead of 12.58%. These results indicate that the combined overhead from instrumentation and fuzzer-side logic is relatively modest. The higher overhead observed in FuzzBench can be attributed to its generally

Table 7. Coverage comparison between different k values (k=1, 10, 100, 1000, 10000, and adaptive) across FuzzBench and UniBench targets. Each number represents the average edge coverage from 10 eight-hour independent runs. Glo_Win shows the number of targets where each setting achieves the highest average coverage. The highest number for each target is marked in bold.

Targets	1	10	100	1K	10K	Ada	Targets	1	10	100	1K	10K	Ada
curl	13,832	13,840	13,900	14,059	13,414	13,935	woff2	2,409	2,383	2,396	2,405	2,334	2,404
bloaty	3,088	3,120	3,050	3,086	3,103	3,093	jhead	357	357	358	360	357	360
harfbuzz	22,675	22,820	22,024	22,244	22,404	22,671	jq	3,378	3,348	3,351	3,352	3,335	3,451
zlib	877	877	877	877	877	877	sqlite3	12,835	12,995	11,849	13,096	12,642	12,574
jsoncpp	1,337	1,331	1,328	1,332	1,333	1,333	cflow	2,073	2,073	2,071	2,072	2,072	2,097
lcms	1,226	1,588	1,643	1,485	1,259	1,683	exiv2	2,103	2,233	2,116	2,106	3,236	2,662
libjpeg	3,208	3,213	3,154	3,071	3,214	3,198	ffmpeg	7,327	9,129	7,606	6,855	6,813	13,933
libpcap	47	47	42	44	47	49	infotocap	1,672	1,695	1,672	1,668	1,656	1,718
libpng	1,405	1,366	1,360	1,361	1,361	1,360	nm-new	1,505	1,567	1,514	1,538	1,543	1,965
libxml2	8,808	8,658	9,044	8,781	8,848	9,546	objdump	4,944	4,990	5,099	4,997	4,990	5,215
libxslt	4,681	4,732	4,742	4,645	4,639	4,659	pdftotext	6,239	6,252	6,256	6,279	6,229	6,267
openh264	13,640	13,608	13,148	13,647	13,628	13,591	mp3gain	1,429	1,415	1,412	1,406	1,453	1,424
openssl	4,576	4,554	4,550	4,569	4,550	4,559	tcpdump	2,916	3,029	3,283	2,998	3,082	4,294
re2	6,255	6,240	6,223	6,269	6,259	6,259	mujs	5,117	4,939	5,039	4,900	4,914	5,456
stbi	2,224	2,221	2,164	2,332	1,931	2,086	tiffsplit	1,096	1,355	1,293	1,201	1,206	1,412
sql	11,774	11,727	11,899	11,616	11,745	11,932	mp4aac	1,615	1,676	1,667	1,662	1,682	1,587
systemd	2,511	2,522	2,521	2,516	2,518	2,522	flvmeta	442	445	442	446	446	436
vorbis	2,042	2,055	2,049	2,049	2,043	2,037	Glo_Win	5	5	2	9	7	16
lame	5,639	5,654	5,653	5,611	5,682	5,679							

Table 8. Mean number of executions (in thousands) of ZTaint-Havoc (Z-H) and AFL++ across FuzzBench and UniBench targets over 10 24-hour independent runs. Overhead shows the percentage reduction in executions of ZTaint-Havoc compared to AFL++ for each dataset.

Targets	Z-H	AFL++	Targets	Z-H	AFL++	Targets	Z-H	AFL++
FuzzBench Targets			sql	88,947	110,594	tcpdump	8,193	8,129
bloaty	31,982	29,848	systemd	24,597	27,272	jq	2,309	2,659
zlib	12,451	12,346	jsoncpp	104,963	135,009	sqlite3	1,896	1,901
lcms	215,976	306,414	libjpeg	651,025	708,923	cflow	3,666	3,686
libpcap	89,014	92,063	libpng	1,059,813	800,547	exiv2	4,769	5,828
libxml2	229,220	249,728	openssl	737,275	925,637	ffmpeg	877	1,124
openh264	3,998	3,997	Overhead	3.84%		infotocap	4,614	4,408
libxslt	497,771	530,097		jhead	12,860	12,642		
re2	123,943	221,797	UniBench Targets			nm-new	6,446	7,183
stbi	35,938	44,769	objdump	5,947	6,110	tiffsplit	13,499	14,177
vorbis	273,842	304,640	mp42aac	9,063	9,835	pdftotext	4,328	4,417
woff2	305,926	444,382	lame	2,498	2,419	mp3gain	7,933	7,473
curl	514,058	711,715	mujs	8,685	8,894	Overhead	12.58%	
harfbuzz	332,316	411,216	flvmeta	16,235	15,883			

higher execution throughput. Since FuzzBench targets typically achieve more executions per second, the fuzzer-side tracking operations become more prominent, leading to increased relative overhead. Nevertheless, this level of overhead remains acceptable for practical fuzzing applications.

Result 7: ZTaint-Havoc shows a runtime overhead of 3.84% on UniBench and 12.58% on FuzzBench, demonstrating that the additional computational cost remains within an acceptable range.

Table 9. **Mean edge coverage of ZTaint-Havoc with solver (ZH+S) and CMPLOG across FuzzBench and UniBench targets over 10 24-hour independent runs. Gain shows the percentage improvement in coverage of ZTaint-Havoc with solver compared to CMPLOG for each dataset. The highest number for each target is marked in bold. * indicates statistically significant differences.**

Targets	ZH+S	CMPLOG	Targets	ZH+S	CMPLOG	Targets	ZH+S	CMPLOG
FuzzBench Targets			sql	11,797	17,111*	tcpdump	7,046*	2,767
bloaty	4,465*	2,798	systemd	2,491	2,488	jq	3,538	2,879
zlib	903	907	jsoncpp	1,341	1,334	sqlite3	15,092	15,415
lcms	1,565	2,684*	libjpeg	3,231	3,231	cflow	2,122*	2,071
libpcap	4,782*	3,573	libpng	2,401	2,691*	exiv2	5,391	5,063
libxml2	13,496*	9,246	libxslt	5,338	5,263	ffmpeg	25,626*	18,305
openh264	13,689	13,522	Gain		3.06%	infotocap	1,890	2,284*
openssl	4,604*	4,320	UniBench Targets			jhead	395*	361
re2	6,268*	6,249	tiffsplit	1,348	1,787*	nm-new	1,706	1,411
stbi	2,798	3,024*	mp42aac	1,607	1,783	objdump	5,811*	4,954
vorbis	2,039	2,041	lame	5722	5544	pdfotext	6,835	6,806
woff2	2,425	2,551*	mujs	6,008	5,974	mp3gain	1,494*	1,416
curl	14,765	14,824*	flvmeta	458	456	Gain		13.54%
harfbuzz	25,483*	24,169						

8.10 RQ8: Comparison with CMPLOG

8.10.1 Experimental Setup. ZTaint-Havoc is designed to enhance the foundational mutation algorithm - havoc mode without any branch-specific solver support. To facilitate a fair comparison with branch-specific approaches like CMPLOG, we implement ZTaint-Havoc with solver, a variant of ZTaint-Havoc equipped with a simple gradient descent-based branch solver, similar to previous work such as [6, 11, 20, 32, 41, 56]. ZTaint-Havoc with solver incorporates specialized branch handling similar to CMPLOG's approach. Using all targets from FuzzBench and UniBench datasets, we conduct 10 independent 24-hour runs with the same seeds as Section 8.4. The remaining experimental settings follow those described in Section 8.7.

8.10.2 Observation. On FuzzBench targets, ZTaint-Havoc with solver outperforms CMPLOG on 10 out of 19 programs, achieving a mean coverage gain of 3.06%. A notable example is bloaty, where ZTaint-Havoc with solver demonstrates a significant 59.6% improvement over CMPLOG. For UniBench targets, ZTaint-Havoc with solver shows superior performance over CMPLOG on 13 out of 17 programs, with an average improvement of 13.54% across all targets. A remarkable case is tcpdump, where ZTaint-Havoc with solver achieves 154.61% improvement in coverage compared to CMPLOG. However, on certain targets like lcms, CMPLOG maintains better performance. This can be attributed to our solver implementation being a simple prototype, lacking some specialized handling mechanisms present in CMPLOG, such as floating-point number processing and string comparisons where both operands are variables. This does not indicate a design limitation of zero-execution FTI, and we plan to incorporate these features in future work to further enhance our solver's capabilities.

Result 8: ZTaint-Havoc with solver demonstrates competitive performance against CMPLOG with average improvements of 13.54% on UniBench and 3.06% on FuzzBench targets.

9 Related Work

AFL[58] ranks among the most prominent fuzzers globally, employing coverage-guided fuzzing to efficiently detect a multitude of vulnerabilities. Numerous researchers have based their subsequent

studies on AFL's approach. AFLFast[9], for instance, incorporates a strategic power schedule to modify both seed selection and the frequency of seed executions. Several other studies[40, 44, 48, 61] have also concentrated on optimizing seed schedules. Additionally, various efforts have been made to tackle constraints that arise during fuzzing, such as utilizing symbolic execution[24, 37, 42, 45, 57] and its variants, including grey-box concolic testing[14], which combines elements of symbolic execution and grey-box fuzzing. Furthermore, certain insightful algorithms[6, 11, 12, 38, 41] have also been proposed to improve constraint solving and fuzzing efficiency.

Havoc is often integrated into many fuzzing processes. MOPT[35] applies a tailored Particle Swarm Optimization to manage the havoc operator. DARWIN[25], as proposed by Patrick et al., employs an Evolution Strategy to systematically refine and adjust the mutation operators' probability distribution. Additional experimentation by Wu et al.[51] has demonstrated the efficacy of Havoc and its synergistic interactions. At SBFT 2025 fuzzing competition[4], our team built an ensemble fuzzer-HFuzz[54] using an early version prototype of ZTaint. This implementation achieved second place in the competition.

Furthermore, Marcel et al. introduced directed fuzzing in AFLGo[8], leveraging the control flow graph (CFG) to compute distances. Recent advancements in directed fuzzing include approaches like Beacon[22], Titan[23], MC²[39] and DAFL[26].

Fuzzing has demonstrated its effectiveness across diverse domains. In software testing[29, 30], it has been instrumental in identifying vulnerabilities across various applications, with applications extending to code similarity analysis[46]. The blockchain domain has seen extensive application of fuzzing techniques for smart contract testing[13, 15, 43, 50, 52]. Most recently, fuzzing has emerged as a promising approach in testing large language models, helping identify potential weaknesses and limitations in these AI systems[16, 17, 53, 55, 60].

10 Discussion

Our method, although effective, encounters several limitations. Firstly, it requires feedback, which entails code instrumentation and can introduce overhead, potentially reducing execution speed. This performance impact must be considered against the advantages our method offers in practical scenarios. Additionally, the success of our approach depends on the selected threshold value. Choosing a threshold that is too low may result in undertaint, possibly missing crucial correlations between input bytes and program behavior. Moreover, our use of filters to minimize noise in taint analysis leads to a smaller sample size for fuzzing-driven taint inference. While this filtering enhances the quality of our taint data, it might also decrease the efficiency of the inference process.

11 Conclusion

In conclusion, this paper presents a computational model of havoc mode and demonstrates how it can be leveraged for zero-execution FTI in coverage-guided fuzzing. Our extensive evaluation on FuzzBench and UniBench shows that ZTaint-Havoc achieves 2.97% and 6.12% higher edge coverage respectively compared to AFL++, demonstrating the effectiveness of utilizing existing havoc mutations for FTI without additional execution.

12 Data Availability Statement

Our research artifacts are publicly available at: <https://github.com/Yu3H0/ZTaint-Havoc>

Acknowledgments

We sincerely appreciate the anonymous reviewers for their valuable feedback and guidance. We also thank Kunpeng Zhang, Shuangjie Yao and Qiao Zhang for their helpful comments and discussions that improved the paper.

References

- [1] 2023. SBST'23 Fuzzing Competition (C/C++ Programs) Report. <https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Coverage/index.html>.
- [2] 2024. a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>.
- [3] 2024. Evaluation report of aflpp on the Fuzzbench dataset. (May 28 2024). <https://www.fuzzbench.com/reports/experimental/2024-05-28-aflpp/index.html>.
- [4] 2025. SBST'25 Fuzzing Competition. <https://sbft25.github.io/tools/fuzzing>.
- [5] 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 (Waikiki, Honolulu, HI, USA) (ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/1985793.1985795
- [6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*.
- [7] Dario Asprone, Jonathan Metzman, Abhishek Arya, Giovanni Guizzo, and Federica Sarro. 2022. Comparing Fuzzers on a Level Playing Field with FuzzBench. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 302–311. doi:10.1109/ICST53961.2022.00039
- [8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.
- [9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
- [10] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.
- [11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE.
- [12] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 499–513.
- [13] Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. 2024. Towards Smart Contract Fuzzing on GPU. In *IEEE Symposium on Security and Privacy (SP)*. 1–15.
- [14] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.
- [15] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.
- [16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.
- [17] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
- [18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.
- [19] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065.
- [20] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX security symposium (USENIX Security 20)*. 2577–2594.
- [21] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.
- [22] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.
- [23] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2023. Titan: Efficient Multi-target Directed Greybox Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 59–59.
- [24] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.
- [25] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2022. DARWIN: Survival of the Fittest Fuzzing Mutators. *arXiv preprint arXiv:2210.11783* (2022).
- [26] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

- [27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
- [28] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.
- [29] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1379–1396. <https://www.usenix.org/conference/usenixsecurity23/presentation/li-wen> (artifact evaluated; badges: Available).
- [30] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*. 1645–1659. doi:10.1145/3576915.3623166 (artifact evaluated; badges: Available, Functional, Reproduced).
- [31] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UniBench}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.
- [32] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jianguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1–17. doi:10.1109/SP46214.2022.9833594
- [33] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. 2023. SBFT Tool Competition 2023–Fuzzing Track. *arXiv preprint arXiv:2304.10070* (2023).
- [34] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. 2023. VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1676–1687.
- [35] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.
- [36] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*.
- [37] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.
- [38] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.
- [39] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2595–2609.
- [40] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. *2022 IEEE Symposium on Security and Privacy (SP)* (2022), 2194–2211.
- [41] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. Fox: Coverage-guided fuzzing as online stochastic control. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. 765–779.
- [42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.
- [43] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
- [44] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. (2021).
- [45] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jianguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. 61–64.
- [46] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.
- [47] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.
- [48] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

- [49] Eric W. Weisstein. [n. d.]. Vector Norm. From MathWorld—A Wolfram Web Resource. <https://mathworld.wolfram.com/VectorNorm.html>
- [50] Taiyu Wong, Chao Zhang, Yuandong Ni, Mingsen Luo, HeYing Chen, Yufei Yu, Weilin Li, Xiapu Luo, and Haoyu Wang. 2024. ConFuzz: Towards Large Scale Fuzz Testing of Smart Contracts in Ethereum. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 1691–1700.
- [51] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.
- [52] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [53] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).
- [54] Yuchong Xie, Yu Liu, Zhibo He, Rundong Yang, Jin Wei, and Dongdong She. 2025. HFuzz: Havoc Mode Guided Fuzzing. In *Proceedings of the 18th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*.
- [55] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1174–1186.
- [56] 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.
- [57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.
- [58] Michał Zalewski. [n. d.]. American Fuzz Lop. <https://github.com/google/AFL>.
- [59] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. *arXiv preprint arXiv:2501.19282* (2025).
- [60] Kunpeng Zhang, Shuai Wang, Jitao Han, Xiaogang Zhu, Xian Li, Shaohua Wang, and Sheng Wen. 2024. Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 508–520.
- [61] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. In *Proceedings of the 44th International Conference on Software Engineering*. 1658–1668.
- [62] Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. 2023. SHAPFUZZ: Efficient Fuzzing via Shapley-Guided Byte Selection. *arXiv preprint arXiv:2308.09239* (2023).

Received 2024-10-31; accepted 2025-03-31