electronics

MDPI

Article

SpeedDGF: Path-Guided Byte Energy for Efficient Directed Grey-Box Fuzzing

Zixuan Ding [1], Shaoguang Wu [1,*], Wei You [2], Lingrui Zhu [3] and Yang Zhang [1]

[1] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 051432, China; zhangyang@hebust.edu.cn (Y.Z.)
[2] School of Information, Renmin University of China, Beijing 100872, China
[3] Department of International Business and Management, University of Nottingham Ningbo China, Ningbo 315104, China
* Correspondence: wushaoguang@hebust.edu.cn

Abstract

Directed grey-box fuzzing (DGF) is central to vulnerability reproduction, patch testing, static analysis, and information-flow detection. However, most existing DGF tools rely on largely random mutation strategies and thus produce many ineffective inputs. To address this, we present SpeedDGF, a directed grey-box fuzzer that employs a lightweight per-byte mutation strategy. At runtime, SpeedDGF dynamically identifies path-sensitive bytes—those most influential for reaching target code regions—and assigns them higher byte-energy, increasing their likelihood of selection during mutation. The effectiveness of SpeedDGF was systematically evaluated across 26 real-world vulnerabilities. The experimental results demonstrate that SpeedDGF is 4.63× faster in reaching the target locations and 5.78× more effective in reproducing target vulnerabilities compared with existing fuzzing tools. This approach significantly enhances the capability of the fuzzing tool to reach specified target locations and efficiently reproduce vulnerabilities.

Keywords: directed grey-box fuzzing; software security; byte energy scheduling

1. Introduction

Directed grey-box fuzzing (DGF) is a widely adopted technique for automated vulnerability discovery and verification. By targeting and exercising specific regions of a program, DGF can more effectively expose potential security flaws, thereby improving both the accuracy and efficiency of vulnerability detection. Compared with traditional fuzzing approaches, DGF enables developers to focus on particular program behaviors, making it possible to uncover vulnerabilities that might otherwise be overlooked. It plays an important role in several critical security applications, including patch testing [1], bug reproduction [2], validation of static analysis results [3], and information-flow monitoring [4].

The key to reproducing specific vulnerabilities is to generate inputs that reach the target region as quickly as possible. This requires mutation algorithms capable of selectively guiding test cases toward those locations. In recent years, considerable effort has been devoted to improving the efficiency of directed grey-box fuzzing (DGF). Most existing techniques leverage structural distances in a program's control-flow or data-flow graph to evaluate and prioritize seed inputs [5,6]. Yet, despite these advances, they often continue to rely on Havoc—a fully random byte-level mutation strategy. Such reliance on unguided mutation significantly constrains the performance and resource efficiency of DGF.

Check for updates

Academic Editor: George Angelos Papadopoulos

Received: 17 January 2026
Revised: 15 February 2026
Accepted: 20 February 2026
Published: 24 February 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

Recent advances in coverage-guided fuzzing have underscored the importance of critical input bytes and introduced techniques to prioritize them for improved fuzzing efficiency. Most existing approaches rely on symbolic-based solutions [7,8] or taint-based mutation methods [9,10] to address complex path constraints that mutation alone cannot easily satisfy. However, both strategies depend on sophisticated symbolic analysis and taint-tracking, which incur substantial runtime overhead. In addition, they require extra instrumentation and program analysis, limiting their practicality and scalability in large real-world scenarios.

To address these limitations, we propose SpeedDGF, a novel directed grey-box fuzzer that introduces a lightweight per-byte mutation strategy. This approach dynamically identifies path-sensitive bytes, which are critical to reaching target code regions, and assigns them a higher "byte-energy," thereby increasing their probability of being selected during mutation. SpeedDGF was empirically evaluated against 26 real-world vulnerabilities that had been previously investigated in the literature. The results demonstrate that SpeedDGF outperforms state-of-the-art fuzzers, including AFL, AFLGo, Windranger, and DAFL, achieving a 4.63× improvement in reaching target locations and a 5.78× improvement in reproducing known vulnerabilities.

The main contributions of this paper can be summarized as follows:

- We propose a lightweight per-byte mutation method with byte-energy analysis and scheduling, which effectively identifies bytes sensitive to paths and target data flows, and prioritizes them in the mutation process.
- We present empirical evaluation results demonstrating that SpeedDGF outperforms state-of-the-art in both reaching target locations and reproducing known vulnerabilities.

2. Motivation Example

To quantitatively verify the inefficiency of unguided mutation, we employed a running example to illustrate our motivation.

Figure 1 presents a seed input of libjpeg-turbo. The content of this seed is displayed in hexadecimal format, with the field types (e.g., Magic Number, File Size, Width, Height, Image Data) labeled above the corresponding byte sequences. Among the total 62 bytes of the seed, 24 bytes (highlighted with a black background) are identified as path-sensitive bytes that directly contribute to guiding the fuzzer toward the target code locations. We conducted 24-h fuzzing experiments targeting the CVE-2018-14498 vulnerability in libjpeg-turbo using state-of-the-art directed grey-box fuzzers, including AFLGo [5], Windranger [6], DAFL [11], and Beacon [12]. Among these, DAFL was able to reach the target locations in approximately 3 h. We further analyzed the test cases generated by DAFL and found that over 60% of the mutations during the early stages of fuzzing are ineffective in progressing toward the target code. Furthermore, as the number of image data bytes increases, the proportion of ineffective mutations is likely to grow, further diminishing fuzzing efficiency. Building upon DAFL, we manually increased the mutation frequency of the critical (highlighted) bytes and observed that the target locations were reached in just 20 min, representing a performance improvement of up to 900%. This result strongly suggests that precisely mutating path-sensitive bytes can significantly improve the efficiency of directed fuzzing.

Our motivating experiment confirms that prioritizing path-sensitive bytes boosts fuzzing efficiency. However, two key challenges remain unresolved. ① How can path-sensitive bytes within an input be identified automatically and dynamically during the mutation phase? ② How can limited mutation energy be efficiently prioritized and allocated to these path-sensitive bytes during the mutation process?

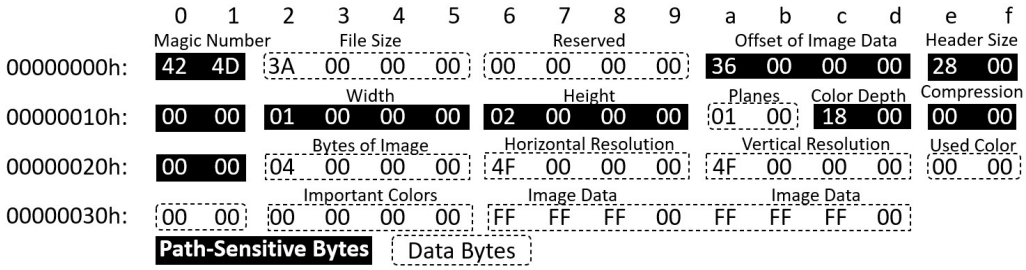


Figure 1. Sample libjpeg-turbo seed.

3. Design

The architecture of SpeedDGF is presented in Figure 2. To address the challenges raised in the motivation, SpeedDGF determines the sensitivity of seed bytes to paths and target data flows through a lightweight per-byte mutation process. Based on these sensitivity metrics, it performs a byte-energy analysis to assign higher byte energy to the most sensitive bytes, thereby increasing the probability of their selection by subsequent byte-energy mutation algorithms. This approach enhances the capability to reach target locations and reproduce target vulnerabilities.

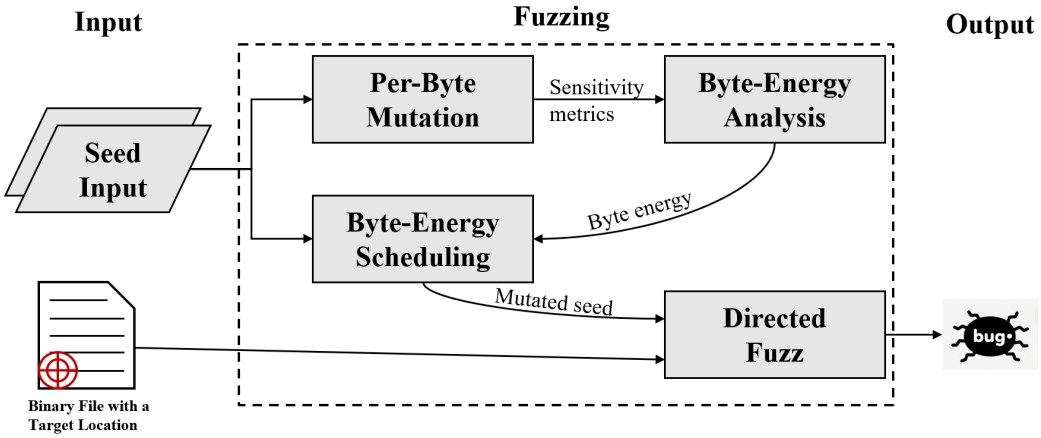


Figure 2. Overview of SpeedDGF.

3.1. Per-Byte Mutation

In fuzzing, each byte of the input seed may influence the program execution flow or variables. When a new seed is fuzzed for the first time, SpeedDGF employs the byte-wise mutations method to analyze the impact of individual bytes on program execution. Initially, it performs four lightweight operations per byte: full flip, LSB flip, +10 increment, and −10 decrement.

For each byte position i in the input seed S, SpeedDGF performs lightweight quadruple mutation operations $\mathcal{M} = m_1, m_2, m_3, m_4$. Each mutation generates a new input $S_i^{m_j}$ and collects its runtime characteristics:

- Execution path hash $H\left(S_i^{m_j}\right)$: A hash value identifying the execution path of the mutated seed;
- Data flow relevance score $D\left(S_i^{m_j}\right)$: Quantifies the ability to cover critical data flow nodes of the target line of code. We adopt the semantic score from prior work DAFL [11] to define this value, where a higher score indicates coverage of more or more critical data flow nodes.

Based on the collected runtime characteristics $H\left(S_i^{m_j}\right)$, $D\left(S_i^{m_j}\right)$, we need to perform variable transformation. These transformed variables should be able to accurately reflect the importance of bytes:

- We verify whether the four mutations maintain identical execution paths. Given the execution path hash $H\left(S_i^{m_j}\right)$, the path consistency flag A_i is:

$$A_i = \begin{cases} 1 & \text{if } H(S_i^{m_1}) = H(S_i^{m_2}) = H(S_i^{m_3}) = H(S_i^{m_4}) \\ 0 & \text{otherwise} \end{cases}$$

- We record the number of mutations causing execution path deviations from the original one. Given the execution path hash $H\left(S_i^{m_j}\right)$, the path perturbation count C_i is:

$$C_i = \sum_{j=1}^{4} \left[H(S_i^{m_j}) \neq H(S)\right]$$

- We record the number of occurrences where the data flow relevance score shows improvement. Given the data flow relevance score $D\left(S_i^{m_j}\right)$, the data flow gain count I_i is:

$$I_i = \sum_{j=1}^{4} \left[D(S_i^{m_j}) > D(S)\right]$$

When the seed file is excessively large, even lightweight byte-wise mutations reduce fuzzing efficiency. In the subsequent evaluation section, DAFL showed strong performance, but SpeedDGF suffered from high mutation overhead due to the oversized seed. To address this, we restrict analysis to the top-m highest-energy bytes (selected once via a *quickselect* algorithm with $O(m)$ complexity), thereby limiting subsequent analyses to $O(m)$ time. Since $m \ll n$, this alleviates performance overhead.

Although per-byte probing introduces an upfront cost proportional to the seed size, this cost is incurred infrequently and only when a seed is first analyzed. In Section 4, we quantify the runtime overhead of this phase and show that it accounts for a small fraction of the total fuzzing time. We further evaluate a top-m configuration to demonstrate that the probing cost can be effectively bounded without significantly affecting effectiveness.

3.2. Byte-Energy Analysis

After obtaining the information from the per-byte mutation phase, it needs to be analyzed to derive byte-energy for subsequent use in byte-energy mutation during the fuzzing stage. During the pre-byte mutation phase, SpeedDGF collects two categories of critical information: the first consists of metrics reflecting a byte's ability to discover new paths, including the Path Consistency Flag A_i and Path Perturbation Count C_i; the second comprises metrics evaluating a byte's capacity to cover more critical data flow nodes, specifically the Data Flow Gain Count I_i. Based on sensitivity metrics from the per-byte mutation phase (A_i, C_i, I_i), we decompose the byte energy into two orthogonal dimensions.

- We evaluate a byte's ability to influence the execution path, focusing on its behavior after multiple mutations. If the byte's execution path hash remains entirely consistent after four mutations, and all these mutations deviate from the original path (i.e., $A_i = 1, C_i = 4$), this indicates that the byte may resemble variables such as magic bytes, whose values must adhere to a strict range. If such bytes are modified arbitrarily, the seed may deviate from the original execution path, thereby failing to pass rigorous checks in the early stages of program execution. Therefore, we assign a lower energy value to such bytes. In other cases, the greater the number of mutations that deviate from the original path (i.e., the higher the value of C_i), the more sensitive the byte appears to be to the execution path. Given the path perturbation count C_i, we define

the path sensitivity energy $E_p(i)$ as follows, where α and β are coefficients, and in this paper we set $\alpha = 10$, $\beta = 20$.

$$E_p(i) = \begin{cases} \alpha & \text{if } A_i = 1 \wedge C_i = 4 \\ \beta * C_i & \text{otherwise} \end{cases}$$

- We evaluate the potential of a byte to cover the target data flow by examining its behavior after multiple mutations. If the byte undergoes four mutations and the number of times it covers increasingly important data flow increases (i.e., the higher the value of I_i), our intuition suggests that this byte is more sensitive to the target data flow. Given the data flow gain count I_i, the data flow sensitivity energy $E_d(i)$ is defined as follows, where γ is a coefficient, and in this paper we set $\gamma = 20$.

$$E_d(i) = \gamma * I_i$$

- We perform a linear weighted fusion of $E_p(i)$ and $E_d(i)$ to achieve collaborative optimization between path exploration and data flow coverage. Given $E_p(i)$ and $E_d(i)$, the final byte energy $E(i)$ is defined as follows, where λ is the weight, and $\lambda = 0.5$.

$$E(i) = \lambda \cdot E_p(i) + (1 - \lambda) \cdot E_d(i)$$

3.3. Byte-Energy Scheduling

SpeedDGF prioritizes the selection of mutated bytes that are more likely to explore new code areas and can cover a greater number (or more critical subsets) of target data flow nodes, rather than selecting all bytes with equal probability. SpeedDGF adopts an energy-driven probability selection mechanism to guide mutation, in order to improve the efficiency of generating inputs. Given the byte energy $E(i)$ of the i-th byte, the mutation probability $P(i)$ for each byte i is defined as follows, where L is the byte length of the seed.

$$P(i) = \frac{E(i)}{\sum_{k=1}^{L} E(k)}$$

It is worth noting that the alias method is employed to optimize the selection of mutation byte positions, reducing the time complexity to O(1).

3.4. Implementation

We implement a prototype SpeedDGF based on DAFL [11], which integrates a data flow graph-based selective instrumentation mode and AFL's built-in random mutation algorithm. We mainly added a lightweight per-byte mutation, byte-energy analysis, and a byte-energy scheduling system in the fuzzing process.

4. Evaluation

4.1. Setup

All experiments were conducted on a workstation equipped with an 8-core 3.6 GHz Intel Core i7-7700 CPU and 8GB of main memory, running the Ubuntu 22.04.5 LTS operating system.

SpeedDGF was compared with the five publicly available advanced fuzzing tools shown in Table 1, which cover two categories of grey-box fuzzers: coverage-based fuzzers (AFL and Angora) and directed grey-box fuzzers (AFLGo, Windranger, and DAFL). Among them, Angora adopts taint-guided targeted mutation, while all the other four tools leverage the classic Havoc mutation strategy. Unfortunately, since these fuzzing tools are not publicly available, it is not possible to compare them with Hawkeye [13], CAFL [14],

and Halo [15]. The authors of Halo were contacted, and they mentioned in an email that the prototype will be released in the future.

Table 1. Comparison Tools.

Fuzzer	Category	Mutation Strategy	Release Version
AFL	Coverage	Havoc mutation	v2.57b
Angora	Coverage	Taint-guided targeted mutation	commit 6b46c85
AFLGo	Directed	Havoc mutation	commit b170fad
Windranger	Directed	Havoc mutation	Docker SHA256 hash 8614ceb
DAFL	Directed	Havoc mutation	commit a6fcc56

To evaluate the performance of SpeedDGF, its capabilities were assessed using various security vulnerabilities in C language programs. A comprehensive dataset was constructed for this purpose. Specifically, vulnerabilities from the CVE datasets included in recent fuzzing tools [11,12,16] were utilized, along with test vulnerabilities from AFLGo [5]. The resulting dataset comprises a total of 26 different types of vulnerabilities, as detailed in Table 2. The dataset does not cover all possible vulnerabilities, mainly due to two reasons: the difficulty in building setups involving complex dependencies, and the lack of critical information in some CVE records themselves. In the fuzzing process, all tasks were allocated 24 h of runtime. Each experiment was repeated five times, and the median value was taken to minimize the impact of fluctuations caused by randomness in fuzzing.

Table 2. Benchmark Composition.

Project	Program	Version	CVE	Type
Ming	Swftophp	0.4.7	2016-9827	Buffer Overflow
			2016-9829	Null Dereference
			2016-9831	Buffer Overflow
			2017-9988	Null Dereference
			2017-11728	Null Dereference
			2017-11729	Null Dereference
			2017-7578	Buffer Overflow
		0.4.8	2018-11095	Use-After-Free
			2018-11225	Buffer Overflow
			2018-20427	Null Dereference
			2019-12982	Buffer Overflow
Lrzip	lrzip	0.631	2018-11496	Use-After-Free
Bintuils	cxxfilt	2.6	2016-4487	Null Dereference
			2016-4489	Use-After-Free
			2016-4490	Integer Overflow
			2016-4492	Integer Overflow
	objcopy	2.8	2017-8393	Buffer Overflow
			2017-8394	Null Dereference
			2017-8395	Null Dereference
	objdump	2.8	2017-8392	Null Dereference
			2017-8397	Buffer Overflow
			2017-8398	Buffer Overflow
	strip	2.7	2017-7303	Null Dereference
Libxml2	xmllint	2.9.4	2017-5969	Null Dereference
			2017-9048	Buffer Overflow
Libjpeg	cjpeg	1.5.90	2018-14498	Buffer Overflow
Total	9		26	4

Unless otherwise stated, SpeedDGF was evaluated without enabling the top-m restriction, in order to present its upper-bound effectiveness. We additionally report the results of a representative top-m configuration to assess scalability.

4.2. Research Questions and Results

We evaluated the effectiveness of SpeedDGF to address the following research questions (RQ).

RQ1: How effective is SpeedDGF in reaching the target line of code?

RQ2: How effective is SpeedDGF in reproducing the target vulnerability?

RQ3: What impact do path-sensitive byte energy and data-flow-sensitive byte energy have on the performance of SpeedDGF?

RQ4: Is the per-byte mutation phase sufficient to accurately identify path-sensitive bytes in real-world programs?

4.2.1. Results for RQ1

To answer RQ1, we employ Time-to-Target (TTT) to evaluate the performance of SpeedDGF against the baseline fuzzers listed in Table 1.

The initial seeds for our tests were directly adopted from the seed set used by DAFL. From this set, we selected 15 examples for which the initial seeds could not reach the target locations, and present the results in Table 3. Cases where the initial seeds already reached the target locations were excluded from our selection. If a fuzzer failed to reach the target line in more than half of the runs within 24 h, we marked the result as T.O. because the median was not available. The best-performing data points are highlighted in bold in the table. For most examples (12 out of 15), SpeedDGF outperformed other fuzzers and achieved the shortest TTT. For performance comparison, we focused on the examples where all fuzzers successfully reached the target line within the 24-h window. On these examples, SpeedDGF achieved a TTT that was 4.2×, 3.77×, 9.11×, and 1.45× faster than AFL, AFLGo, Windranger, and DAFL, respectively. In fact, Angora performs even worse than AFL. In the few vulnerabilities where neither our tool nor Angora timed out, our tool is 15.67× faster. The reason lies in its coverage-guided approach, which attempts to solve all program constraints, leading to excessive overhead and thus low efficiency.

Table 3. Results of TTT. The data represents the seconds required to reach the target line of code.

Program	CVE	Line	AFL	Angora	AFLGo	WindRanger	DAFL	*SpeedDGF*
swftophp	2016-9827	outputscript.c:1687	79	1395	68	82	68	**61**
	2016-9829	parser.c:1656	443	T.O.	354	5770	166	**67**
	2017-9988	parser.c:2995	1532	T.O.	1525	1463	1342	**656**
	2017-11728	decompile.c:1699	118	T.O.	104	113	30	**25**
	2017-11729	decompile.c:1440	119	T.O.	99	72	42	**26**
	2018-11095	decompile.c:1843	3538	4864	2298	808	518	**422**
	2018-11225	decompile.c:2015	28,180	T.O.	T.O.	T.O.	12,815	**12,450**
	2018-20427	decompile.c:425	233	T.O.	138	77	20	**18**
	2019-12982	decompile.c:3120	**15**	T.O.	55	41	21	25
cxxfilt	2016-4487	cplus-dem.c:4319	15	T.O.	30	**11**	15	**11**
	2016-4489	cplus-dem.c:3007	**4**	T.O.	10	6	5	5
	2016-4490	cp-demangle.c:1596	25	T.O.	51	26	25	**21**
objcopy	2017-8394	stream.c:1747	841	1223	922	1064	300	**102**
objdump	2017-8397	reloc.c:885	37	T.O.	46	37	**30**	41
cjpeg	2018-14498	rdbmp.c:209	T.O.	T.O.	T.O.	T.O.	10,307	**2465**
Best perf.			2	0	0	1	1	12

The results indicate that SpeedDGF exhibits superior capability in reaching the designated target lines compared to the baseline fuzzers.

4.2.2. Results for RQ2

To answer RQ2, we use Time-to-Exposure (TTE) to compare the performance of SpeedDGF against all fuzzers shown in Table 1.

The experimental results on all 26 examples are illustrated in Table 4, where T.O. represents a timeout if a fuzzer failed to reproduce the target vulnerability in more than half of the runs within 24 h. The best-performing data points are highlighted in bold in the table. For most examples (20 out of 26), SpeedDGF outperformed other fuzzers and achieved the shortest TTE. For performance comparison, we focused on the examples where all fuzzers successfully reproduced the target vulnerability within the 24-h window. For these examples, SpeedDGF achieved TTEs that were 5.33×, 5.55×, 10.03×, and 2.21× faster than those of AFL, AFLGo, Windranger, and DAFL, respectively. In fact, Angora performs worse than AFL, while our tool is 29.45× faster. The reason lies in its coverage-guided approach, which attempts to solve all program constraints, leading to excessive overhead and thus low efficiency.

Table 4. TTE on programs from benchmark vulnerabilities. The data represents the seconds required to trigger the vulnerability.

Program	CVE	AFL	Angora	AFLGo	Windranger	DAFL	*SpeedDGF*
Swftophp	2016-9827	79	1395	68	82	68	**61**
	2016-9829	443	T.O.	354	5770	166	**67**
	2016-9831	709	10,793	599	T.O.	249	**201**
	2017-9988	1532	T.O.	1525	1463	1342	**656**
	2017-11728	2024	T.O.	1531	2513	328	**177**
	2017-11729	966	T.O.	573	1371	215	**71**
	2017-7578	227	T.O.	266	T.O.	**103**	161
	2018-11095	3538	4864	2298	808	518	**422**
	2018-11225	28,180	T.O.	T.O.	T.O.	12,815	**12,450**
	2018-20427	**2999**	T.O.	3287	5692	6760	3320
	2019-12982	T.O.	T.O.	57,966	T.O.	6923	**3466**
lrzip	2018-11496	10	**2**	12	10	14	8
cxxfilt	2016-4487	477	T.O.	464	435	235	**141**
	2016-4489	1099	T.O.	1200	**310**	886	491
	2016-4490	288	T.O.	228	166	59	**38**
	2016-4492	1505	T.O.	5292	3074	735	**271**
objcopy	2017-8393	2581	3906	2146	2049	838	**733**
	2017-8394	841	1223	922	1064	300	**102**
	2017-8395	141	997	126	108	**16**	39
objdump	2017-8392	357	1617	469	T.O.	54	**31**
	2017-8397	56,503	T.O.	67,694	T.O.	47,369	**38,419**
	2017-8398	1634	10,110	2095	823	1248	**126**
strip	2017-7303	146	**15**	161	3204	330	554
xmllint	2017-5969	326	T.O.	375	557	178	**118**
	2017-9048	T.O.	T.O.	T.O.	T.O.	3051	**2072**
cjpeg	2018-14498	T.O.	T.O.	T.O.	T.O.	10,313	**2467**
$\overline{\text{CV}}$		0.84	NA	0.91	0.77	0.93	0.83
Best perf.		1	2	0	1	2	20

To evaluate the stability of multiple experimental results, we adopt the Coefficient of Variation (CV) to measure the relative dispersion of data, which reflects the fluctuation of multiple runs relative to the mean value. For each fuzzer, we compute the CV only on CVE vulnerabilities that do not time out in all tests across all fuzzers, and use its mean $\overline{CV}$ to represent the overall stability across repeated experiments. Since Angora has an excessive number of timeouts, we exclude it from comparison. The results demonstrate that SpeedDGF is only slightly weaker than Windranger in terms of stability. However, SpeedDGF significantly outperforms Windranger in the most critical metric: vulnerability detection efficiency.

We further evaluated two variants of SpeedDGF, SpeedDGF-top1000 (with the top-m byte limit enabled, m = 1000) and SpeedDGF-NoBE (byte energy optimization disabled while retaining byte-by-byte probing capability)—on three cases with large-scale input seeds (CVE-2017-8393, CVE-2017-8397, CVE-2018-14498). Experimental results are shown in Table 5. For SpeedDGF-top1000, only minor changes in TTE (Time-To-Exposure) were observed, and the overall trend remains largely consistent with that of full byte-by-byte analysis. SpeedDGF-NoBE preserves the byte-by-byte probing approach for all bytes; compared to ordinary DAFL, it merely adds a per-seed byte-by-byte probing step before mutation, and its performance overhead remains within an acceptable range.

Table 5. Impact of Byte Limitation (Top-m) and Energy Optimization Strategies on SpeedDGF Performance.

Program	CVE	SpeedDGF-top1000		SpeedDGF-NoBE	
		TTE	Speedup (SpeedDGF)	TTE	Speedup (DAFL)
objcopy	2017-8393	679	1.08×	1009	0.83×
objdump	2017-8397	36,223	1.06×	53,248	0.89×
cjpeg	2018-14498	2590	0.95×	10,796	0.96×

Experimental results demonstrate that SpeedDGF is more effective than the baseline fuzzers when reproducing target vulnerabilities.

4.2.3. Results for RQ3

To answer RQ3, we investigate how path-sensitive byte energy and data-flow-sensitive byte energy affect the performance of SpeedDGF. To this end, we instantiate two variants of SpeedDGF: (1) SpeedDGF-PathSen, which sets the fusion weight to $\lambda = 1$ and relies exclusively on path-sensitive energy to guide byte mutation; and (2) SpeedDGF-DataSen, which sets the fusion weight to $\lambda = 0$ and uses only data-flow-sensitive energy to guide byte mutation. We evaluate and compare the performance of SpeedDGF, its variants, and DAFL across all 26 vulnerabilities, as shown in Table 2.

Figure 3 presents the results. Overall, using each type of byte energy in isolation provides clear guidance benefits over DAFL, and their combination consistently yields better results than using either one alone. In terms of vulnerability reproduction, SpeedDGF-PathSen and SpeedDGF-DataSen are, respectively, 1.48× and 1.27× faster than DAFL. Notably, even in cases where an individual type of byte energy fails to improve the performance of DAFL (e.g., SpeedDGF-PathSen and SpeedDGF-DataSen on CVE-2017-8397), the simultaneous use of both byte energies still enables SpeedDGF to outperform DAFL overall. These results demonstrate the synergistic effect between the two types of byte energy: when combined, SpeedDGF reproduces vulnerabilities on average 2.05× faster than DAFL.

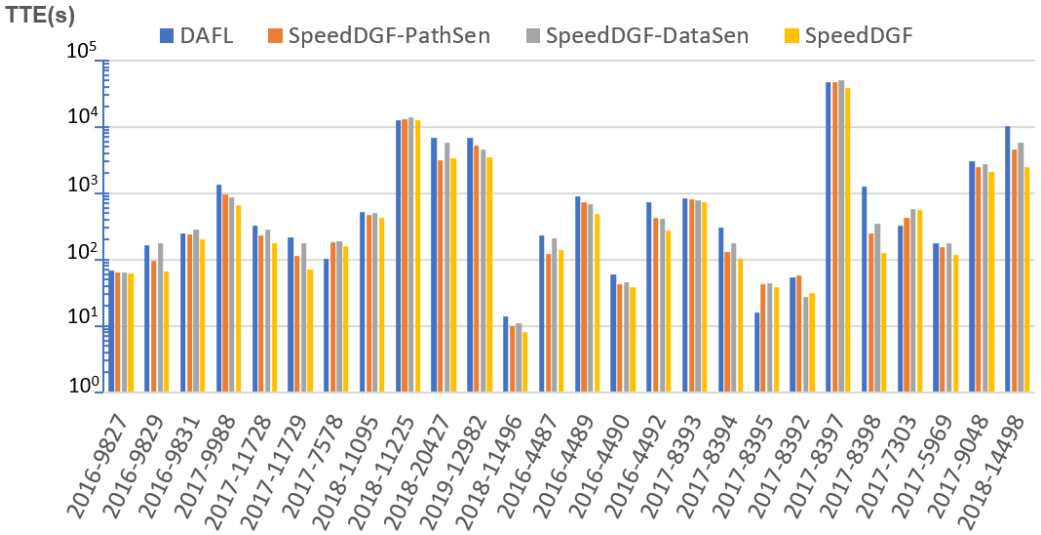


Figure 3. Impact of Path-Sensitive and Data-Flow-Sensitive Byte Energy on the Performance of SpeedDGF (TTE).

4.2.4. Results for RQ4

The design of SpeedDGF relies on a lightweight per-byte probing strategy, which estimates the sensitivity of individual input bytes to program execution paths and target-related data flows by performing four byte-level mutations per byte. The primary motivation behind this design is to reduce analysis overhead and improve overall fuzzing efficiency. Nevertheless, it remains necessary to evaluate to what extent such a lightweight probing approach can accurately characterize the true sensitivity of input bytes. Based on this consideration, we design a set of dedicated experiments to assess the accuracy of the proposed "four-mutation" byte probing strategy. Specifically, we first manually inspect how the application processes individual input bytes to distinguish path-sensitive bytes and construct a reference ground truth. We then compare the path-sensitive bytes identified by SpeedDGF against this ground truth, thereby measuring SpeedDGF's false positive rate and false negative rate.

Table 6 presents the detailed experimental results. As shown, the proposed probing strategy achieves relatively low false positive and false negative rates in detecting path-sensitive bytes, with average values of 8.65% and 8.32%, respectively. Combined with the ablation results in RQ3, these findings indicate that even when a small number of misclassifications occur during the per-byte probing phase, the synergistic effect of the lightweight per-byte probing strategy in the subsequent mutation guidance process can effectively mitigate their impact, thereby improving the overall efficiency of SpeedDGF's directed exploration.

Table 6. Accuracy Evaluation of SpeedDGF's Per-Byte Mutation Strategy in Identifying Path-Sensitive Bytes.

Program	Actual	Inferred	Wrong (FP *)	Missed (FN **)
cjpeg	24	26	2 (7.69%)	0 (0%)
libtiff	52	49	3 (6.12%)	6 (11.54%)
lrzip	21	22	2 (9.09%)	1 (4.76%)
mupdf	70	74	11 (14.86%)	7 (10%)
xmllint	85	77	5 (5.49%)	13 (15.29%)

* FP = #Wrong/#Inferred ** FN = #Missed/#Actual.

Overall, the experimental results demonstrate that the four-mutation per-byte probing strategy is capable of effectively distinguishing path-sensitive bytes in real-world programs

with low overhead, providing a reliable foundation for subsequent byte-energy assignment and directed mutation.

5. Related Works

5.1. Coverage-Guided Grey-Box Fuzzing

Coverage-guided greybox fuzzing (CGF) aims to maximize code coverage in order to discover hidden bugs. Here, the widely used tool AFL [17] is taken as a representative example to illustrate the principles of CGF. AFL employs lightweight instrumentation techniques at compile time to capture basic block transitions and obtains coverage information at runtime. It then selects a seed from the seed queue and mutates this seed to generate test cases. If a test case executes a new path, it is added to the queue as a new seed. AFL prefers seeds that trigger new paths and assigns them higher priority than non-preferred seeds. Compared with other instrumentation-based fuzzers, AFL incurs a moderate performance overhead.

Since 2013, research on coverage-guided greybox fuzzing has increased steadily, and the application domains of CGF have continuously expanded. For example, it has been applied to JavaScript engines [18,19], binary testing [20,21], blockchain smart contracts [22], operating system kernels [23,24], and more. In addition to AFL, many easy-to-use and efficient fuzzing tools have emerged, such as libFuzzer [25], Honggfuzz [26], and Syzkaller [27]. Among them, ProFuzzer [28] adopts specialized mutation strategies for different types of fields, thereby reducing the search space; MemFuzz [29] performs coverage testing using simple memory read/write instructions, such as load and store; and TortoiseFuzz [30] focuses on error-prone code regions, such as system call functions and loop structures. Some tools are designed to provide feedback information for fine-grained exploration of program states. For instance, PathAFL [31] introduces the hpath to explore new program execution paths, while Angora [10] incorporates n-gram-based code control-flow context information. In addition, Memlock [32] monitors peak memory consumption and is limited to discovering potential memory-consumption-related vulnerabilities, such as recursion bugs.

5.2. Directed White-Box Fuzzing

Directed whitebox fuzzing tools are mainly integrated into symbolic execution engines, such as KLEE [33], KATCH [34], and BugRedux [2]. Directed Symbolic Execution (DSE) leverages program analysis and constraint solving to generate inputs that can systematically and effectively explore the state space of feasible paths [35]. Once a target path is identified, test cases are generated to explore potential solutions to the path constraints. Since most paths are actually infeasible, the search usually proceeds iteratively by finding feasible paths to intermediate targets. Unlike directed fuzzing, which treats the reachability problem as an optimization problem that minimizes the distance between generated seeds and their targets [5], DSE treats reachability as an iterative constraint satisfaction problem. DSE is effective in a variety of scenarios, such as reaching error-prone program locations (e.g., critical system calls [36]), testing code patches [37,38], executing corner-case paths to increase coverage [36], and reproducing failures internally [39,40].

However, the effectiveness of DSE comes at the cost of efficiency. The heavyweight program analysis and constraint solving used by DSE are time-consuming. In each iteration, DSE employs program analysis techniques to identify branches that can be negated to get closer to the target. It then constructs the corresponding path conditions based on the instruction sequences along these paths. Finally, it uses a constraint solver to check the satisfiability of these conditions. Compared with DSE, directed fuzzing can generate far more inputs within a given time budget. Böhme et al. demonstrated experimentally that directed fuzzing outperforms DSE in both effectiveness and efficiency. For example,

AFLGo was able to expose the Heartbleed vulnerability within 20 min, whereas the DSE tool KATCH failed to do so even after 24 h [5].

5.3. Directed Grey-Box Fuzzing

Existing directed grey-box fuzzing approaches primarily focus on optimization from two perspectives: first, optimizing seed selection strategies; second, optimizing directed grey-box fuzzing by avoiding irrelevant executions.

Currently, most research efforts are dedicated to improving seed selection strategies. AFLGo [5] is one of the earliest tools to implement the concept of directed grey-box fuzzing. Its technical principle is as follows: it first calculates the average weighted distance between execution paths and target basic blocks on the control-flow graph (CFG). Based on this, it assigns higher mutation priorities to seeds with shorter weighted distances. Consequently, the tool can more effectively guide the testing process toward target code regions. Subsequently, several research teams have proposed further optimizations based on AFLGo. For example, Hawkeye [13] and WindRanger [6] enhance the accuracy of target distance calculations by introducing more informative feedback mechanisms. FuzzGuard [16] employs deep learning models to filter out irrelevant test cases, thereby improving the guidance efficiency of AFLGo. TargetFuzz [41] approaches the problem from the perspective of seed corpus construction, enhancing testing effectiveness by building target-oriented seed sets. ParmeSan [42] not only retains distance feedback but also incorporates sanitizer-guided vulnerability coverage analysis, expanding the dimensions of guidance information. In contrast to the above methods, DAFL [11] introduces a data-flow semantics-aware distance evaluation method to address the inaccuracies in distance calculation that often occur in traditional control flow-based fuzzing during indirect calls.

Some directed grey-box fuzzing tools have made significant contributions to avoiding irrelevant executions. For instance, DAFL [11] introduces selective instrumentation technology, focusing on analyzing code regions related to target locations. Beacon [12] and SieveFuzz [43] modify the program under test to prune irrelevant paths during runtime. FuzzGuard [16] also utilizes deep learning to filter out irrelevant test cases, effectively enhancing AFLGo.

5.4. LLM-Assisted Program Analysis and Vulnerability Detection

As software systems grow increasingly complex, traditional program analysis techniques have shown limitations in scalability and semantic modeling. Recently, large language models (LLMs) have attracted growing attention in program analysis and software security due to their strong capabilities in code understanding and contextual reasoning.

Recent surveys systematically review LLM-assisted program analysis and classify existing approaches into static, dynamic, and hybrid analysis, covering tasks such as vulnerability detection, program comprehension, symbolic execution assistance, and test generation [44]. These studies highlight the potential of LLMs while also pointing out persistent challenges in analysis precision, interpretability, and accurately modeling real program execution.

In vulnerability detection, recent empirical studies evaluate advanced LLMs primarily on Java programs, with additional experiments on C/C++ to assess cross-language generalization [45]. Results show that although some models achieve strong performance under specific configurations, detection effectiveness varies significantly across models, programming languages, and learning paradigms, and can be comparable to random guessing in certain settings. Expanding the context window generally improves performance, whereas other factors have more limited impact.

Overall, existing research indicates that while LLMs show promise for automated vulnerability detection and program analysis, their effectiveness is highly scenario- and configuration-dependent, with notable limitations in robustness and execution-aware reasoning. Consequently, LLMs are typically viewed as complementary to traditional program analysis and testing techniques rather than standalone solutions. In contrast, fuzzing explores program behaviors through concrete execution and coverage feedback, retaining advantages in discovering execution-dependent vulnerabilities. How to effectively integrate LLMs with coverage-guided fuzzing remains an open research direction.

6. Conclusions

This paper proposes a directed grey-box fuzzing tool called SpeedDGF, which employs a lightweight per-byte mutation to perform byte-wise mutations and identify path-sensitive bytes. These critical bytes are assigned higher byte-energy, increasing their likelihood of being selected by subsequent mutation algorithms. The experimental results demonstrate that, in reaching target locations, SpeedDGF is 4.63× faster than state-of-the-art fuzzing tools; in reproducing existing vulnerabilities, it is 5.78× faster than state-of-the-art fuzzing tools. This approach enhances the ability of the directed grey-box fuzzing tool to reach target locations and reproduce vulnerabilities efficiently.

Author Contributions: Conceptualization, Z.D., Y.Z., W.Y., S.W. and L.Z.; methodology, Z.D.; implementation, Z.D.; validation, Z.D., Y.Z. and W.Y.; formal analysis, Z.D.; investigation, Z.D. and W.Y.; resources, Y.Z.; data curation, Z.D.; writing—original draft preparation, Z.D.; writing—review and editing, Z.D., W.Y., Y.Z. and S.W.; supervision, Y.Z. and W.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work is partially supported by the pre-research Hebei Natural Science Foundation under Grant No. F2026208004, and the Hebei Overseas High-level Talent Foundation under Grant No. C20230358.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source code is publicly available at https://github.com/Hyman111/SpeedDGF (accessed on 17 January 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Böhme, M.; Oliveira, B.C.d.S.; Roychoudhury, A. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*; ACM: New York, NY, USA, 2013; pp. 334–344.
2. Jin, W.; Orso, A. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2012; pp. 474–484.
3. Christakis, M.; Müller, P.; Wüstholz, V. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*; ACM: New York, NY, USA, 2016; pp. 144–155.
4. Mathis, B.; Avdiienko, V.; Soremekun, E.O.; Böhme, M.; Zeller, A. Detecting information flow by mutating input data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: New York, NY, USA, 2017; pp. 263–273.
5. Böhme, M.; Pham, V.-T.; Nguyen, M.-D.; Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*; ACM: New York, NY, USA, 2017; pp. 2329–2344.
6. Du, Z.; Li, Y.; Liu, Y.; Mao, B. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*; ACM: New York, NY, USA, 2022; pp. 2440–2451.
7. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*; San Diego, CA, USA, 21–24 February 2016; Volume 16, pp. 1–16.

8. Zhao, L.; Duan, Y.; Xuan, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–27 February 2019.
9. Wang, T.; Wei, T.; Gu, G.; Zou, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*; IEEE: New York, NY, USA, 2010; pp. 497–512.
10. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2018; pp. 711–725.
11. Kim, T.E.; Choi, J.; Heo, K.; Cha, S.K. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*; USENIX: Berkeley, CA, USA, 2023; pp. 4931–4948.
12. Huang, H.; Guo, Y.; Shi, Q.; Yao, P.; Wu, R.; Zhang, C. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2022; pp. 36–50.
13. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 2018*; ACM: New York, NY, USA, 2018; pp. 2095–2108.
14. Lee, G.; Shim, W.; Lee, B. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 2021*; USENIX: Berkeley, CA, USA, 2021; pp. 3559–3576.
15. Huang, H.; Zhou, A.; Payer, M.; Zhang, C. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2024*; IEEE: New York, NY, USA, 2024; pp. 1956–1973.
16. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*; USENIX: Berkeley, CA, USA, 2020; pp. 2255–2269.
17. Google. American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 17 January 2026).
18. Park, S.; Xu, W.; Yun, I.; Jang, D.; Kim, T. Fuzzing javascript engines with aspect-preserving mutation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020*; IEEE: New York, NY, USA, 2020; pp. 1629–1642.
19. He, X.; Xie, X.; Li, Y.; Sun, J.; Li, F.; Zou, W.; Liu, Y.; Yu, L.; Zhou, J.; Shi, W.; et al. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 2021*; ACM: New York, NY, USA, 2021; pp. 2229–2242.
20. Nagy, S.; Nguyen-Tuong, A.; Hiser, J.D.; Davidson, J.W.; Hicks, M. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 2021*; ACM: New York, NY, USA, 2021; pp. 351–365.
21. Gao, X.; Duck, G.J.; Roychoudhury, A. Scalable fuzzing of program binaries with E9AFL. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2021*; IEEE: New York, NY, USA, 2021; pp. 1247–1251.
22. Wüstholz, V.; Christakis, M. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 2020*; ACM: New York, NY, USA, 2020; pp. 1398–1409.
23. Xu, M.; Kashyap, S.; Zhao, H.; Kim, T. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020*; IEEE: New York, NY, USA, 2020; pp. 1643–1660.
24. Pailoor, S.; Aday, A.; Jana, S. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In P*roceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 2018*; USENIX: Berkeley, CA, USA, 2018; pp. 729–743.
25. Google. Libfuzzer—A Library for Coverage-Guided Fuzztesting. Available online: https://github.com/llvm-mirror/llvm/blob/master/docs/LibFuzzer.rst (accessed on 17 January 2026).
26. Google. Honggfuzz. Available online: https://github.com/google/honggfuzz/ (accessed on 17 January 2026).
27. Google. Syzkaller-Kernel Fuzzer. Available online: https://github.com/google/syzkaller/ (accessed on 17 January 2026).
28. You, W.; Wang, X.; Ma, S.; Huang, J.; Zhang, X.; Wang, X.; Liang, B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019*; IEEE: New York, NY, USA, 2019; pp. 769–786.
29. Coppik, N.; Schwahn, O.; Suri, N. MemFuzz: Using memory accesses to guide fuzzing. In *Proceedings of the 12th IEEE Conf on Software Testing, Validation and Verification (ICST), Xi'an, China, 2019*; IEEE: New York, NY, USA, 2019; pp. 48–58.
30. Wang, Y.; Jia, X.; Liu, Y.; Zeng, K.; Bao, T.; Wu, D.; Su, P. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Proceedings of the 27th Network and Distributed System Security Symp (NDSS 2020), San Diego, CA, USA, 23–26 February 2020*; NDSS: San Diego, CA, USA, 2020; pp. 1–12. Available online: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24422-paper.pdf (accessed on 28 November 2022).
31. Yan, S.; Wu, C.; Li, H.; Shao, W.; Jia, C. PathAFL: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conf on Computer and Communications Security, Taipei, Taiwan, 2020*; ACM: New York, NY, USA, 2020; pp. 598–609.

32. Wen, C.; Wang, H.; Li, Y.; Qin, S.; Liu, Y.; Xu, Z.; Chen, H.; Xie, X.; Pu, G.; Liu, T. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd ACM/IEEE Int Conf on Software Engineering, Seoul, Republic of Korea, 2020*; ACM: New York, NY, USA, 2020; pp. 765–777.
33. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, 2008*; USENIX: Berkeley, CA, USA, 2008; pp. 209–224.
34. Marinescu, P.D.; Cadar, C. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 2013*; ACM: New York, NY, USA, 2013; pp. 235–245.
35. Ma, K.K.; Yit Phang, K.; Foster, J.S.; Hicks, M. Directed symbolic execution. In *Proceedings of the International Static Analysis Symposium, Venice, Italy, 2011*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 95–111.
36. Heartbleed Bug. Available online: https://heartbleed.com/ (accessed on 5 May 2024).
37. Clang Static Analyzer. Available online: https://clang-analyzer.llvm.org/ (accessed on 3 May 2024).
38. Santelices, R.A.; Chittimalli, P.K.; Apiwattanapong, T.; Orso, A.; Harrold, M.J. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 2008*; IEEE: New York, NY, USA, 2008; pp. 218–227.
39. Cousot, P.; Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79), San Antonio, TX, USA, 1979*; ACM Press: New York, NY, USA, 1979; pp. 269–282.
40. Rößler, J.; Zeller, A.; Fraser, G.; Zamfir, C.; Candea, G. Reconstructing core dumps. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 2013*; IEEE: New York, NY, USA, 2013; pp. 114–123.
41. Canakci, S.; Matyunin, N.; Graffi, K.; Joshi, A.; Egele, M. Targetfuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Seoul, Republic of Korea, 2022*; ACM: New York, NY, USA, 2022; pp. 561–573.
42. Österlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. ParmeSan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 2020*; USENIX: Berkeley, CA, USA, 2020; pp. 2289–2306.
43. Srivastava, P.; Nagy, S.; Hicks, M.; Bianchi, A.; Payer, M. One fuzz does not fit all: Optimizing directed fuzzing via target-tailored program state restriction. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC), Austin, TX, USA, 2022*; ACM: New York, NY, USA, 2022; pp. 388–399.
44. Wang, J.; Ni, T.; Lee, W.B.; Zhao, Q. A Contemporary Survey of Large Language Model Assisted Program Analysis. *arXiv* **2025**, arXiv:2502.18474. [CrossRef]
45. Lin, J.; Mohaisen, D. From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection. In Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–28 February 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.