FrameShift: Resizing Fuzzer Inputs Without Breaking Them

Harrison Green
harrisog@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Claire Le Goues
clegoues@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Fraser Brown
fraserb@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Coverage-guided fuzzers are powerful automated bug-finding tools. They mutate program inputs, observe coverage, and save any input that hits an unexplored path for future mutation. Unfortunately, without knowledge of input formats—for example, the relationship between formats' data fields and sizes—fuzzers are prone to generate destructive *frameshift* mutations. These time-wasting mutations yield malformed inputs that are rejected by the target program. To avoid such breaking mutations, this paper proposes a novel, lightweight technique that preserves the structure of inputs during mutation by detecting and using *relation fields*.

Our technique, FrameShift, is simple, fast, and does not require additional instrumentation beyond standard coverage feedback. We implement our technique in two state-of-the-art fuzzers, AFL++ and LibAFL, and perform a 12+ CPU-year fuzzer evaluation, finding that FrameShift improves the performance of the fuzzer in each configuration, sometimes increasing coverage by more than 50%. Furthermore, through a series of case studies, we show that our technique is versatile enough to find important structural relationships in a variety of formats, even generalizing beyond C/C++ targets to both Rust and Python.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

fuzzing, structure-inference

ACM Reference Format:
Harrison Green, Claire Le Goues, and Fraser Brown. 2026. FrameShift: Resizing Fuzzer Inputs Without Breaking Them. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3787765

1 Introduction

Fuzzing is an effective tool for exploring the state space of programs and finding bugs. While the earliest fuzzer simply fed random data into UNIX programs [31] (and found bugs!), coverage-guided fuzzers like AFL++ [13], and LibAFL [14] use mutations and feedback to explore targets more efficiently. These coverage-guided fuzzers maintain a growing corpus of inputs. They pick inputs from this corpus, apply random mutations (bit-flips, arithmetic operations, insertions from a dictionary, etc.), and then measure feedback like edge or block coverage. The fuzzers retain mutated inputs that reach new coverage, and discard mutated inputs that don't.

ICSE '26, Rio de Janeiro, Brazil
2026. ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787765

These fuzzers are quite versatile and widely adopted in industry: coverage-guided fuzzers have found more than 13,000 vulnerabilities across 1,000+ open-source projects as part of Google's OSS-Fuzz.[1]

Unfortunately, even the best modern fuzzers struggle to successfully mutate certain types of input structures. Many common fuzz targets operate over serialized binary formats whose *metadata*—e.g., `size` and `offset` fields—describes the layout of associated data buffers. Security-critical applications process such structured inputs: Hardware security chips, for example, operate on `TPM` packets, and `openssl` (and others) use `DER`-encoded `ASN.1` messages; both formats contain multiple nested size fields, which make them notoriously hard to mutate [3].

In general, these sorts of fields occur in almost every serialized binary format—in codecs (`PNG`, `JPEG`, `MP3`, `OGG`, etc.), document formats (`PDF`, `XLSX`, `DOCX`, etc.), cryptographic protocols (`TLS`, `SSH`), object formats (`ELF`, `PE`, `Mach-O`), and many more. Fundamentally, any multi-part, variable-length binary data format *requires* metadata to describe its structure and delineate field boundaries.

These metadata-rich formats pose a challenge to modern fuzzers. When a fuzzer mutates specific parts of an input—like a variable sized data buffer—without correspondingly updating related parts of the input—like the size or offset fields describing that buffer—it renders the input structurally invalid. We call such destructive mutations *frameshifts*. As a result, the target program will mishandle the input or abort early with a validation error. Frameshifts cause fuzzers to get stuck exploring invalid inputs and inputs with the same sized structures as the seed corpus, unable to discover inputs with resized or shifted data.

Existing approaches to this problem either (1) augment the fuzzer with an input specification, allowing it to understand and generate the expected structure [11, 32], or (2) learn important structures automatically during fuzzing using e.g., static analysis [10, 22], coverage-guided feedback [12, 44], or (recently) a combination of static analysis and machine learning [36].

Unfortunately, techniques in the first category require manual effort and risk over-constraining the fuzzer. Techniques in the second category also fall short: AIFORE [36] and ProFuzzer [44] are closed source, and all of TIFF [22], WEIZZ [12], ProFuzzer, and AIFORE focus on discerning byte types (integer, enum, string, etc.) rather than identifying the relationship between size fields and their target buffers—and thus cannot perform validity-preserving resizing mutations.

Further, modern fuzzers include a (rough) generalization of the techniques prior work uses to e.g., discern valid `enum` options, byte ranges, etc. AFL++ and LibAFL, for example, use sophisticated strategies like compare-logging (an adaptation of RedQueen [2]),

[1] https://github.com/google/oss-fuzz

which attempts to find special values and inject them into the input. This type of technique can, for example, identify alternative `enum` options (by instrumenting switch cases), or required magic bytes (by instrumenting e.g. `memcmp`), thus subsuming less general analyses that do (some of) the same thing. Unfortunately, these features are not always enabled by default, rendering them absent from some academic evaluations [10, 36]—and potentially underselling the actual performance of modern general-purpose fuzzers. In our work, we use industry-standard configurations (following FuzzBench [30]) and focus on designing a system that confers an actual, significant benefit over state-of-the-art baseline fuzzers, even in their optimal configurations.

This paper presents a new approach to fuzzing structured input formats by discovering relation fields and using them for structure-aware resizing mutations that preserve input validity. Our approach, FrameShift, is built on two key insights. First, coverage loss between a seed and a mutated input indicates that the fuzzer *may* have mutated an important relation field (e.g., size field) without mutating the corresponding data. This indicates a *potential* destructive frameshift that can be identified dynamically, over the course of a coverage campaign. There are, however, other reasons—reasons beyond frameshifts—that a mutation can lead to coverage loss. For example, mutations to enums may redirect execution to a different code path, or mutations to checksums may cause the target to abort early. Thus, our second insight is that, to identify *true* frameshifts, FrameShift can conduct experiments to find points where resizing a buffer restores coverage with respect to the original destructive mutation. We prune the search space of potential corrective mutations using domain-specific heuristics that let the analysis run in mere seconds per new input. Finally, FrameShift uses its newly-discovered relations to inform fuzzing with existing mutators—thus doing structure-aware fuzzing that avoids destructive frameshifts.

FrameShift is designed to be fast, easy to integrate, and compatible with modern fuzzers: inspired by prior work [12, 44], it requires no instrumentation beyond coverage feedback and no manual format specifications. On unfriendly targets, it incurs minimal overhead; on friendly targets, it helps achieve new coverage quickly.

Contributions. We show that FrameShift is:

- **Effective** compared to industry-leading, state-of-the-art fuzzers. It increases coverage by an average of 6%—and more than 50% in certain configurations—while only suffering a 5.5% coverage loss (on average) for the worst-case target.
- **Versatile**. Unlike static analysis-based approaches, it is not limited to C/C++: we run FrameShift out-of-the-box on Rust and Python fuzz harnesses.
- **Capable** of identifying real, target-specific size and offset fields, even when nested, in a variety of popular binary formats. It even discovers semantic differences in input formats across two programs parsing the same input type.

Our implementations [2] [3] and evaluation framework [4] are available open-source.

[2] https://github.com/hgarrereyn/AFLplusplus-FrameShift
[3] https://github.com/hgarrereyn/LibAFL-FrameShift
[4] https://github.com/hgarrereyn/frameshift-eval

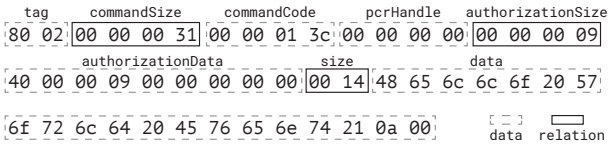


Figure 1: Example, annotated `TPM_PCR_Event` packet.

```
ExecuteCommand(uint32_t reqSize, char *req)
 1 COMMAND cmd = {
 2   .paramSize = reqSize, .paramBuffer = req
 3 };
 4 if (!Parse_U16(&cmd.tag, &cmd))          goto Err;
 5 if (!Parse_U32(&cmd.cmdSize, &cmd))      goto Err;
 6 if (!Parse_U32(&cmd.cmdCode, &cmd))      goto Err;
(1)7 if (cmd.cmdSize != reqSize)            goto Err;
 8
 9 if (cmd.tag == TPM_ST_SESSIONS) {
10   if (!Parse_U32(&cmd.authSize, &cmd)) goto Err;
(2)11  if (cmd.authSize > cmd.paramSize)    goto Err;
12   if (!ParseSessionBuffer(&cmd))        goto Err;
13 } else {
14   if (!CheckAuthNoSession(&cmd))        goto Err;
15 }
16
17 switch (cmd.cmdCode) {
18   case TPM_CC_PCR_Event:
19     PCR_Event event;
(3)20    if (!Parse_Event(&event, &cmd))     goto Err;
21     TPM2_PCR_Event(&event);
22     break;
23 }
```

Figure 2: Simplified example parsing logic for a `TPM_PCR_Event` in `ms-tpm-20-ref`; note the three size validation checks.

2 Overview

This section describes the challenges that fuzzers face with structured inputs (Section 2.1), and then outlines the intuition behind our approach (Section 2.2). We use the `TPM` format as a running example by investigating the `ms-tpm-20-ref` target,[5] a reference TPM 2.0 specification and simulator developed by Microsoft.

2.1 Motivating Example

Figure 1 shows a **`TPM_PCR_Event`** command packet (command code **`0x13c`**) with the payload "Hello World Event!". The parsing and command execution logic (Figure 2, simplified for presentation) contains a number of interesting edge cases for a fuzzer to discover. It parses fields from the data buffer (lines 4-7); validates authorization data, or follows an alternative path (**`CheckAuthNoSession`**) if there exists no authorization session (lines 9-15); and then invokes the correct execution handler (lines 17-22). If any of the parser checks fail, execution terminates with error handling (`goto Err`).

Ideally, a fuzzer will reach the call to **`TPM2_PCR_Event`** (line 21) with many different inputs. To do so, a fuzzer must navigate three nested size fields that must stay synchronized with one another and with the described data: **`cmdSize`** must match the total packet size (❶); **`authSize`** must fit within the remaining data (❷); and payload

[5] https://github.com/microsoft/ms-tpm-20-ref

Fuzzer	Corpus Size	❶	❷	❸
State-of-the-art Fuzzers				
AFL++ [13]	17062	41	36	0
LibAFL [14]	23360	52	53	0
AFL [45]	15694	35	38	0
WEIZZ [12]	24231	16	15	0
NestFuzz [10]	11593	28	32	0
AFL++(FS)	21693	359	399	14
LibAFL(FS)	37255	194	287	8

Table 1: Newly-sized variants passing highlighted validation checks in `ms-tpm-20-ref` with (bottom) and without (top) FrameShift (FS).

event **`size`** (from the payload data, interpreted as a variable size **`PCR_Event`** buffer) must equal the size of the remaining **`data`** (❸).

It is therefore *extremely difficult* for a fuzzer to mutate an existing valid TPM message into another valid message with a different payload size. A mutation changing the message size must also change **`cmdSize`**; insertions or deletions in **`authData`** must also update **`authsize`**; attempts to resize the **`data`** buffer must also update **`size`**.

To demonstrate the challenge this poses, we ran five state-of-the-art fuzzers (along with our own 2 FrameShift variants) on the `ms-tpm-20-ref` target using the example TPM packet from Figure 1 as the seed input. Each fuzzer ran for 48 hours for 10 repetitions (see subsection 4.1 for the full experimental setup). We then analyzed the resulting corpus to see how frequently each fuzzer was able to find *newly sized* (i.e. differing in **`cmdSize`**, **`authSize`**, or **`size`** from the seed) inputs that passed each of the highlighted validation checks.

Table 1 shows results. *None* of the state-of-the-art fuzzers were able to find a *single* newly sized input that got to—and passed!—check ❸. All generated inputs that *did* pass that check had an **`authSize`** of 9 bytes and a data payload **`size`** of exactly 20 bytes, like the seed. These fuzzers were effectively stuck, unable to successfully perform a resizing mutation even after 20 CPU-days of fuzzing.[6]

Our FrameShift variants, under the same configuration, were able to find 14 and 8 newly sized **`TPM_PCR_Event`** commands respectively, unlocking new codepaths in the **`TPM2_PCR_Event`** handler. Furthermore, they were both able to find an order of magnitude more newly sized inputs reaching the prior checks, discovering many more command types in the process. As a result, FrameShift variants found an average of 15.3% more coverage than the baseline fuzzers on `ms-tpm-20-ref` in a 48-hour fuzzing campaign (§4.3).

This example is illustrative of an issue affecting a wide range of binary formats (`ELF`, `PNG`, `ASN.1`, etc.) that are all essential in the modern software stack.

2.2 FrameShift Intuition

Our technique identifies structural metadata—the position and target of each size field in the TPM input packet—and augments a fuzzer to preserve the relationships between that metadata automatically during mutation. The key idea is to (1) identify *potential* relation fields by observing mutations that cause coverage loss, and to (2) validate *true* relation fields by experimenting with new mutations that restore coverage by changing data size. Such "double-mutants" indicate a likely relation field.

Figure 3 illustrates FrameShift's *double-mutation* experiments on our example **`TPM_PCR_Event`** packet:

2.2.1 Disrupting Coverage (A). The first key observation (also underlying ProFuzzer [44]) is that it is possible to *indirectly* identify validation checks at runtime using loss-of-coverage as evidence. For example, mutating the **`cmdSize`** field in a valid seed will produce a new input that fails to reach some originally-covered program bits (those after check ❶ in Figure 2).

In Figure 3, row Ⓐ highlights the bytes, up to **`size`**, that lose coverage when incremented by 0x20; no bytes in the **`data`** field are highlighted, because mutating them does not change the execution path. The highlighted bytes are *candidate* relation fields that may correspond to metadata in the format that must remain synchronized with input data.

2.2.2 Restoring Coverage (B). Loss of coverage for a candidate relations can be explained by either: (1) a *frameshift* mutation, or (2) some other validated part of the testcase (uninteresting for our purposes). For example, mutating **`cmdCode`** may cause the program to trigger a different command handler.

To disambiguate these scenarios, FrameShift aims to automatically identify points at which bytes can be inserted into the input to restore (most of) the original coverage. If the fuzzer mutated a size field by incrementing its value by N, there should be a point in the input where inserting N bytes "re-syncs" size with data. Critically, this is likely only possible if the original mutated field *actually described a size*. It is unlikely, for example, that a fuzzer can increment a **`cmdCode`** by N, and then insert N bytes elsewhere to restore the original execution behavior. Although there are 29 bytes in the **`TPM_PCR_Event`** packet that disrupt coverage when mutant, there are only three for which we can find an associated *insertion point* that restores some original coverage (Figure 3, ❶, ❷, and ❸).[7]

Row Ⓑ❶, shows that inserting right before the **`pcrHandle`** restores a small percentage of coverage (lightly shaded). More is restored by inserting near the end of the file, preserving the authentication section (passing ❷ and ❸). Row Ⓑ❷ corresponds to insertions that correct the **`authSize`** field; again, the most coverage is restored when inserting after the **`authData`** region. Finally, part Ⓑ❸ corresponds to the **`size`** field of the **`PCR_Event`**, where any insertion inside the **`data`** region restores coverage equally.

This example illustrates the intuition behind FrameShift's approach to dynamically identifying relation fields. Testing every byte or insertion point (as in this illustration) is prohibitive in practice; moreover, discovering certain relation fields is often impossible without discovering others (e.g., **`cmdSize`**). Our implementation uses heuristics to tractably prune the search space and produce a practical analysis (on the order of seconds or milliseconds per input).

3 FrameShift

In this Section, we expand on the intuition from §2 and describe our design in detail. We start by formalizing the concept of a *relation field*, an abstraction over different types of size/offset fields (§3.1). We describe how to discover these relation fields, using heuristics to

[6] WEIZZ and NestFuzz do identify some structures, but not these fields.

[7] The latter bytes are identifiable only after **`cmdSize`** is discovered.

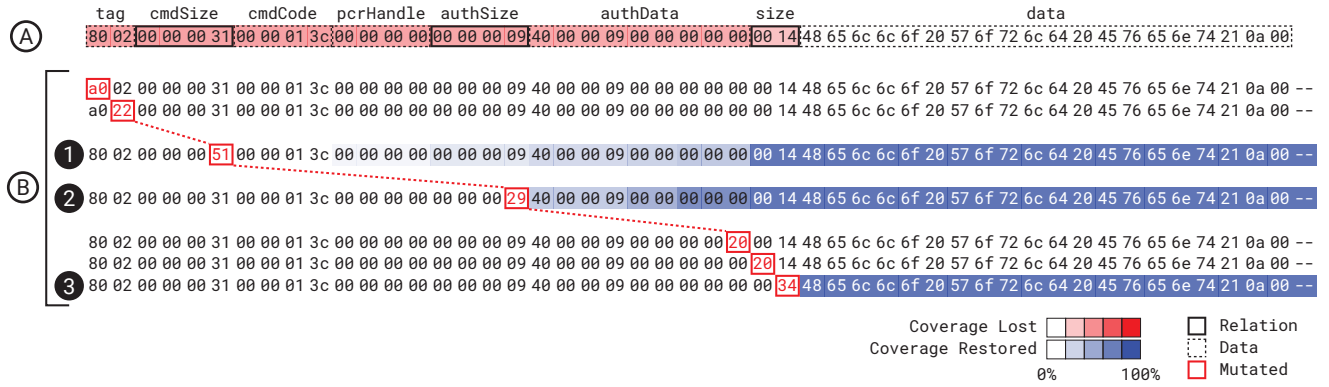


Figure 3: Illustration of the double-mutant experiment on a `TPM_PCR_Event` packet. (A): The original input; bytes that cause a loss-of-coverage when mutated are highlighted red. (B): Each row shows a mutated byte that caused a loss-of-coverage (red box) and every insertion point that restored coverage (blue highlight). Some rows with no coverage restoration are omitted for brevity (dotted red line). Rows 1, 2, and 3 correspond to the discovery of the highlighted validation checks in Figure 2.

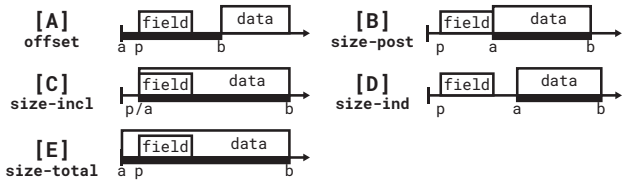


Figure 4: Examples of size/offset fields as relation fields.

prune the search space (§3.2). Finally, we discuss how FrameShift uses relations to implement structure-aware mutations (§3.3).

3.1 Structured Inputs

Disruptive frameshift mutations occur when a fuzzer modifies input bytes that correspond to metadata (e.g. a size field), without fixing up the corresponding data. To avoid frameshifts, we identify size fields and their corresponding data regions. For the purposes of mutation, `size` and `offset` fields can be treated equivalently. A `size` represents the length of some span of the input data. An `offset` represents the distance from the input start to some point in the data; effectively a `size` field for everything before it.

We generalize these as a *relation field* $R := (a, b, p, s, e)$ consisting of a field at position p with size s and endianness e, that represents the length of some span of the input data. The span is defined by a start position a and an end position b, where $a < b$.

This construction generalizes many types of size/offset fields, as depicted in Figure 4. For an offset field (A), the start position a is 0 and the end position b designates where the data starts. For a size field, the actual data field starts at a and ends at b. However, the field itself may be positioned in different ways with respect to the data it describes, i.e., immediately before it (B), just inside it (C), some other arbitrary location (D), or the size field may be the entire input (E).

3.2 Automatic Relation Discovery

Relation analysis discovers important relation fields in inputs and is run once per new corpus input a fuzzer discovers. When the fuzzer attempts to mutate an input associated with relations, it can then use relation data to "fix up" implicated fields.

The analysis first identifies destructive mutations that lead to coverage loss. These indicate that the mutated bytes *might* correspond to relation fields. The analysis then attempts *restorative* mutations to undo the original mutation's destruction. If such restoration is possible, it indicates that the first mutation was indeed a frameshift caused by mutating a relation field. It also provides evidence as to the nature of the relation (i.e., what part of the input it describes).

3.2.1 Destructive and restorative mutations. A given input I's coverage profile F is represented abstractly as a set of bits:

$$F(I) := \{f_1, f_2, \ldots, f_n\}$$

Depending on the instrumentation, these bits may represent blocks hit, edges taken, etc. We set threshold T_{loss} to indicate the percentage of coverage loss necessary to signal a destructive mutation, i.e., I^- is a destructive mutation iff.:

$$|F(I) - F(I^-)| \geq T_{\text{loss}} \cdot |F(I)|$$

Conversely, a *restorative* mutation restores some of the lost coverage, defined by the threshold T_{restore}. Mutating input I^- produces a *restoring mutant* I^+ iff.:

$$|F(I^+) \cap (F(I) - F(I^-))| \geq T_{\text{restore}} \cdot |F(I) - F(I^-)|$$

In our experiments, $T_{\text{loss}} = 0.05$ and $T_{\text{restore}} = 0.2$. These values work well, but per-target tuning is an interesting future direction.

3.2.2 Candidate Relation Field Identification. The first step in the analysis is to identify potential relation fields. The analysis iterates over every field size $s \in \{8, 4, 2, 1\}$, endianness $e \in \{\text{big}, \text{little}\}$ and potential field position $p \in \{0, \ldots, \text{size}(i) - s\}$. For each configuration, the analysis deserializes the input bytes at position p to obtain value v. To prune the search space, it only considers fields with a value $v \leq \text{size}(i)$ for some input i, since sizes or offsets must have values that are at most the size of the input.

The analysis mutates the values of each candidate field to test if doing so is destructive. For $s > 1$, the mutation increments by `0xff`, forcing a carry from the least-significant byte (to distinguish

between little- and big-endian fields); for $s = 1$, the mutation increments the value by `min(0x20, 0xff-v)`, which is enough to cause frameshifts, without overflowing a field. If the mutations lead to coverage loss above T_{loss}, the mutated bytes are a candidate relation, and analysis continues with insertion point discovery.

3.2.3 Insertion Point Discovery. Not all coverage loss is caused by frameshifts. Mutating important constants, checksums, or enum values can also invalidate input or change execution flow. The second analysis phase seeks to validate whether candidates are *true* relation fields, and collects evidence about their nature if so.

For each candidate, the analysis iterates over potential insertion points—places where bytes can be inserted—in search of restorative mutations. Iterating over every byte in the input is prohibitive, as it requires invoking the target program for every byte. We limit the search based on a smaller set of *anchor points*. In practice, most relation fields occur in one of the forms depicted in Figure 4. Thus, the `start` of the target span is often one of 0 (for offset / size-total fields), p (for size-inclusive fields), or $p + s$ (for size-post fields). In each of these cases, we test the corresponding `end` position (`start` + v) as a candidate insertion point. We select the insertion point that restores the most coverage, as long as it exceeds T_{restore}.

The size-indirect form (D) is more challenging: the start position may occur at an arbitrary input point. In practice, since the target program needs to be able to actually locate this data, there is often some other metadata (another size or offset field) that indicates the start of the data. For example, in `ELF` files, there are 8-byte `size` and `offset` fields describing the location of mapped program and section data. To find these fields efficiently, we expand the insertion point search to consider start positions at $R.p$, $R.a$, and $R.b$ for every true relation field R that we have *already* discovered. During analysis, FrameShift first identifies the `offset` field, and then uses it as an anchor point to identify `size` field insertion points.

FrameShift is a fundamentally heuristic analysis: candidate relation fields with viable insertion points are *likely* (but not guaranteed) to be real relation fields. It is possible that inserting bytes can restore coverage for other reasons, e.g., if the initial mutation actually modified an `enum` value, but the inserted bytes coincidentally reintroduced another input structure which hit the original codepath. We rarely observe this in practice.

3.3 Structure-aware Mutation

We now describe how FrameShift adapts standard fuzzer mutations to account for relations, via *structure-aware* mutations.

3.3.1 Raw Mutations. Standard fuzzer mutations manipulate a raw input I three ways: `Replace`(I, i, V) (replace the subsequence starting at i with V), `Insert`(I, i, V) (insert subsequence V at position i), and `Remove`(I, i, n) (remove n bytes after position i). Fuzzer mutators typically perform several of these actions at once. For example, a *splicing* mutator combines `Replace` and `Insert`.

3.3.2 Structured Mutations. We redefine these mutation operators to act on a structured input $S := (I, \mathbb{R})$ with input I and set of learned relations $\mathbb{R}$, by first applying the mutation to the underlying input I and then invoking `OnInsert` or `OnRemove` to track the change and update the associated relation fields. This bookkeeping is required, as fuzzers often perform multiple mutations in sequence. After all mutations, and before executing the test case, FrameShift re-serializes relation fields to apply their new values to the underlying input.

3.3.3 Accommodating Havoc. While FrameShift is designed to identify and preserve resizing mutations, we aim to avoid restricting the fuzzer from making useful destructive mutations that can unlock new coverage. Therefore, during a mutation, if the fuzzer tries to perform an action that is incompatible with the current set of relations—e.g., injecting bytes in the relation field itself (between p and $p + s$)—FrameShift ignores that relation field for the rest of the mutation chain, avoiding re-serializing it. Thus, FrameShift updates relation fields without also over-constraining the fuzzer.

Algorithm 1: OnInsert and OnRemove

OnInsert:
Data: Relation R, index i, sequence V
if $i \leq R.p$ **then** $R.p \leftarrow R.p + |V|$;
if $i < R.a$ **then** $R.a \leftarrow R.a + |V|$;
if $i \leq R.b$ **then** $R.b \leftarrow R.b + |V|$;

OnRemove:
Data: Relation R, index i, size n
if $i \leq R.p$ **then** $R.p \leftarrow R.p - \min(R.p - i, n)$;
if $i \leq R.a$ **then** $R.a \leftarrow R.a - \min(R.a - i, n)$;
if $i \leq R.b$ **then** $R.b \leftarrow R.b - \min(R.b - i, n)$;
return R

4 Evaluation

This section answers the following research questions:

- **RQ1 (Coverage)**: How does FrameShift compare to SOTA binary and structure-aware fuzzers? (Section 4.2)
- **RQ2 (Bug Finding)**: To what extent does FrameShift improve bug finding ability? (Section 4.3)
- **RQ2 (Applicability)**: Where is FrameShift most/least effective? What are the failure cases? (Section 4.3)
- **RQ3 (Case Study)**: Which relations can FrameShift identify in real-world targets? (Section 4.4)
- **RQ4 (Versatility)**: How versatile is FrameShift on different languages and types of coverage feedback? (Section 4.5)

4.1 Experimental Setup

We implement FrameShift's algorithm in both AFL++ and LibAFL, industry-leading fuzzers; they are permissively open source. Our implementations integrate with existing mutators and require no instrumentation changes. These variants are denoted as AFL++(FS) and LibAFL(FS) throughout evaluation.

The AFL++ implementation of FrameShift consists of 600 lines of C that implement a new fuzzer stage to run analysis and store relation metadata in queue inputs. AFL++(FS) tracks insertions and deletions from the havoc and splice mutators, and then re-serializes relation data before executing test cases.

For LibAFL, we write a modular fuzzer stage and custom input type in roughly 1600 lines of Rust. The implementation is functionally identical to the AFL++ fork and canonically implemented, and

Benchmark	Format	Commit
bloaty	ELF/Mach-O/WebAssembly	`52948c1`
freetype2	TTF/OTF/WOFF	`cd02d35`
harfbuzz	TTF/OTF/TTC	`cb47dca`
lcms	ICC-profile	`f0d9632`
libjpeg-turbo	JPEG	`3b19db4`
libpcap	PCAP	`17ff63e`
libpng	PNG	`cd0ea2a`
ms-tpm-20-ref	TPM	`6b72d66`
openh264	H.264	`045aeac`
openssl	DER	`b0593c0`
openthread	IPV6-packet	`2550699`
qpdf	PDF	`2cb2412`
vorbis	OGG	`84c0236`
woff2	WOFF	`8109a2c`
jsoncpp	JSON (text)	`8190e06`
libxml2	XML (text)	`c7260a4`

Type	Fuzzer	Version
Binary	AFL++	`v4.21c`
Binary	LibAFL	`f343376`
Binary	AFL	`v2.57b`
Structured	NestFuzz	`d16eb69`
Structured	WEIZZ	`c9cbeef`

Table 2: Setup: 16 benchmark programs, and 5 fuzzer baselines.

is thus plug-and-play with many LibAFL modules. This lets us use LibAFL's multi-language support out of the box.

4.1.1 Benchmarks. Table 2, top, shows the 16 benchmarks used for the large-scale coverage experiment (RQ1): all binary-format targets, and two text-based formats (`jsoncpp` and `libxml2`) from FuzzBench [30], and two binary formats (`ms-tpm-20-ref` and `qpdf`) from OSS-Fuzz [35], inspired by community discussions about hard-to-fuzz formats [3]. The text-based formats are a worst-case baseline (since they do not contain serialized relation fields).

4.1.2 Baseline Fuzzers. We select five baseline fuzzers (Table 2, bottom) representing the state-of-the-art both in general purpose coverage-guided fuzzing, and automated binary structure-aware fuzzing. AFL++ [13] and LibAFL [14] act as direct baselines for our prototype; AFL [45] is the baseline for NestFuzz.

We also compare to two fuzzers that do structural inference. WEIZZ [12] uses coverage feedback and extra instrumentation to identify structures in chunk-based binary formats. NestFuzz [10] models a program's input processing program via dynamic taint analysis to discover dependencies. We do not include AIFORE [36] or ProFuzzer [44] because both are closed-source.[8] TIFF [22] requires paid decompiler software and relies on now-outdated versions of Intel Pin; recent results suggest it would be outperformed by both NestFuzz and WEIZZ.

[8]Authors of AIFORE did not respond to our request for code. Authors of ProFuzzer did not respond to our clarification question when we could not run their tool.

4.1.3 Configurations and Coverage. For our prototype and all baseline fuzzers except NestFuzz, we use the FuzzBench configuration. In both LibAFL and AFL++, this includes REDQUEEN-style compare-logging [2] and dictionaries, two features that work well in practice [27]. Our tool variants are configured identically to AFL++ and LibAFL, except they include a new FrameShift fuzzer stage, and the ability to fix inputs after resizing. Because NestFuzz does not have a FuzzBench configuration, we use the configuration provided in the project README. We run each corpus through LLVM-instrumented benchmarks to compute total edge coverage, evaluating fuzzer performance. For full details of our evaluation setup, see the provided artifact.

4.1.4 Hardware. We ran the large-scale fuzzing experiment on Google Cloud C3 instances with Intel Sapphire Rapids processors. The bug finding evaluation and case studies ran on dedicated servers with two Intel(R) Xeon(R) Gold 6430 @ 3.40GHz and 1 TB of RAM.

4.2 RQ1: Coverage

To compare FrameShift's performance to state-of-the-art fuzzers, we ran a large-scale fuzzing experiment with 16 benchmarks[9] and 7 fuzzers/fuzzer configurations. Table 3 shows results. We tested fuzz runs from both an empty corpus (E in the results tables; here, ability to learn structure quickly is particularly important) and from a corpus with a single high-quality seed (S in the results tables; here, ability to find seed variants quickly is important). We ran each fuzzer/benchmark/corpus configuration 10 times for 48 hours, reporting resulting arithmetic mean edge coverage.

The last row displays the average score per fuzzer, following FuzzBench conventions: the percentage of maximum coverage obtained (where a fuzzer achieving the highest coverage scores 100, and 70% of maximum scores 70).

4.2.1 Results. Table 3 shows results. For both empty and seeded corpus configurations, the FrameShift variants were most effective, achieving the highest coverage in 10/16 of the benchmarks in both cases. AFL++(FS) achieved the highest average score (97.3) on the empty corpus by more than 7 points, followed by LibAFL(FS) (89.6). In the seeded corpus setting, LibAFL(FS) achieved the highest average score (96.7), followed by LibAFL (95.1) and then AFL++(FS) (94.1). This ordering mirrors the baseline fuzzers themselves, where AFL++ performs (relatively) better from an empty corpus, while LibAFL performs better from a seeded corpus. For both baseline fuzzers and corpus configurations, the introduction of FrameShift significantly increased their performance.

The other three fuzzers (AFL, WEIZZ, and NestFuzz) were generally not competitive, finding the highest coverage on only three benchmarks across both configurations and underperforming the baselines fuzzers. This is likely because our evaluation uses state-of-the-art FuzzBench configurations.

4.3 RQ2: Bug Finding

While coverage is a useful proxy for fuzzer effectiveness (as examined in RQ1) and often correlates with bug finding ability [6], it ultimately matters only when it enables the fuzzer to find bugs. A potential concern, for example, is that by maintaining validity of

[9]NestFuzz on `bloaty` and `openssl` required prohibitive build modifications.

	AFL++(FS)		LibAFL(FS)		AFL++		LibAFL		AFL		WEIZZ		NestFuzz	
Benchmark	E	S	E	S	E	S	E	S	E	S	E	S	E	S
bloaty	2095	2468	**2330**	**4551**	1858	1904	2005	3825	643	3157	732	959	†	†
freetype2	8242	10142	8456	10491	**9441**	**10715**	8985	10317	3796	7572	4301	5093	3719	7284
harfbuzz	6835	**7124**	6661	6810	**7034**	7046	6688	6992	4020	5309	3243	3878	3950	5156
lcms	**1940**	**2093**	1852	2070	1184	1810	1753	2084	1250	1203	1601	1571	36	551
libjpeg	**1739**	2344	587	2288	950	**2367**	510	2318	656	2299	451	1948	478	2316
libpcap	**3041**	**3010**	2623	2846	2835	2678	2468	2689	36	2497	2094	2095	35	2068
libpng	**1943**	**1991**	1798	1965	1860	1963	1804	1962	1549	1936	1270	1690	7	1191
ms-tpm-20-ref	2845	3180	**2889**	**3253**	2209	2696	2685	3047	2409	2868	2072	2281	2029	2223
openh264	8498	8491	8386	8305	8469	8485	8414	8457	8502	**8521**	6976	7178	**8509**	8496
openssl	**4670**	**5178**	4506	4906	4007	4879	4644	4690	4593	4773	3654	4299	†	†
openthread	2424	2505	2614	**3000**	2495	2521	**2654**	2962	2287	2380	2444	2655	2239	2373
qpdf	1181	1999	1157	1851	**1203**	**2228**	1169	1886	1165	1574	947	969	440	1045
vorbis	**954**	1253	509	1239	945	1264	397	1253	205	1257	206	1246	205	**1267**
woff2	**939**	**1060**	934	1043	769	1043	814	1043	7	1003	713	1011	7	990
jsoncpp (text)	**510**	**510**	507	508	509	**510**	507	507	508	508	508	508	508	157
libxml2 (text)	12551	12392	13130	13265	12838	**14069**	**13251**	13200	12366	10934	7442	7460	7107	7289
Average Score	**97.3**	94.1	89.6	**96.7**	88.6	92.1	86.6	95.1	61.7	84.4	63.9	71.9	42.4	68.8

Table 3: Arithmetic mean edge coverage after 48 h (10 runs) for each fuzzer–benchmark pair. Highest average coverage for each benchmark is in bold (for both empty and seeded corpus). E: empty corpus, S: seeded corpus. †: target failed to build.

relation fields, FrameShift may inadvertently *overconstrain* the fuzzer to only valid testcases, thus miss bugs triggered by malformed input. As we show below, this is not the case.

We evaluate FrameShift's impact on bug discovery on the Magma benchmark (v1.2), 138 real-world bugs in 9 programs (4 binary formats and 5 text-based formats) that have been *front-ported* to more recent commits. Each bug requires reaching a certain location in the program and satisfying a trigger condition. While no bug benchmark is perfect, Magma is a commonly used and realistic benchmark recommended by the fuzzing community and best practices [6, 34]. We evaluate both AFL++(FS) and its direct baseline AFL++ to isolate just the effect of FrameShift. We ran each variant on the full suite of programs for 24 hours with ten repetitions.

4.3.1 Results. Counting bugs reached/triggered in *any* of the ten iterations, AFL++ reached 65 bugs and triggered 38, while AFL++(FS) reached 66 bugs (adding `TIF004`, four times) and also triggered 38. Each triggered one unique bug: AFL++(FS) found `SSL009` in `openssl`, while AFL++ found `XML012` in `libxml2`.

On average, however, each *individual* AFL++(FS) run reached and triggered more bugs than AFL++ (Table 5). For binary formats, AFL++(FS) had a statistically significant increase in bugs reached and triggered for `libtiff` and bugs triggered for `openssl` (despite reaching the same number as AFL++). For text-based formats, AFL++(FS) was not significantly better or worse except for `libxml2`, where it found 0.6 bugs fewer on average.

However, total bug counts after 24 hours are largely dominated by bugs found right around the timeout. For bugs which were found quicker by both tools, we analyze the *time-to-trigger* for each bug. Specifically, following recent work [15], we calculate the *hazard ratio* of each bug by fitting a Cox proportional hazards model [7] (note that this is only possible for bugs found by both tools). The hazard ratio correlates with the time-to-trigger *speedup* as a result of FrameShift. Values greater than 1 indicate that AFL++(FS) found the bug quicker than AFL++, while values less than 1 indicate that AFL++ found the bug quicker. Table 6 shows the hazard ratios for each bug when the measure was statistically significant ($p < 0.05$).

The results show that FrameShift confers a significant speedup in almost all cases for binary formats, with the opposite effect (as expected) for text-based formats.

4.4 RQ3: Applicability

To understand the specific contribution of FrameShift over baseline, we visualize final coverage values for each of the 10 fuzzer runs per benchmark (Table 4). Each graphic shows the FrameShift-enabled variant runs (blue lines above the centerline) along with the AFL++ baseline runs runs (red lines below the centerline). The lines are plotted on a linear axis. The left-most side represents the run with the least coverage; the right-most side, the run with the most. For each configuration, we report the average change in coverage when enabling FrameShift (Δ%). Following best practices for fuzzer evaluations [23, 34], we use the Mann-Whitney U-test to compute the statistical significance of differences in fuzzer performance (p column in the Table). Statistically significant results ($p < 0.05$) are shaded green (FrameShift found more coverage) or red (FrameShift found less coverage).

4.4.1 Results. Table 4 shows results comparing to AFL++, which benefits more from FrameShift integration. FrameShift enables a statistically significant increase in coverage in 15/32 configurations, and a decrease in only 6. For LibAFL, the effect is more muted: a statistically significant increase in 6 configurations and a decrease in 4 (results omitted from the Table, for space). In both, the magnitudes of the increase are generally larger than the decreases.

There are 7 benchmarks where FrameShift obtains a statistically significant increase in coverage of at least 3%, 6 benchmarks where it performs roughly neutral, and 3 benchmarks where it has a negative effect. All of the 7 positive benchmarks contain

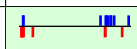
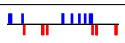
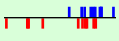
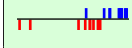
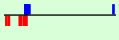
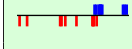
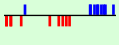
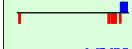
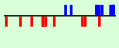
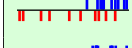
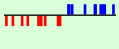
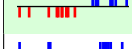
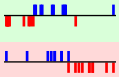
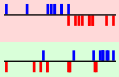
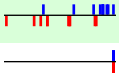
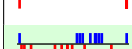
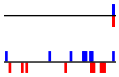
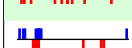
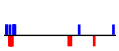
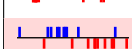
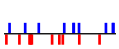
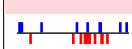
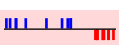
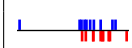
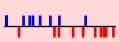
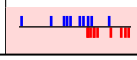
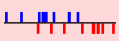
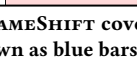
	AFL++(FS) vs. AFL++					
Benchmark	Empty	Δ%	p	Seeded	Δ%	p
libjpeg		+83.0	*		-0.9	
lcms		+63.9	**		+15.6	*
bloaty		+12.7	**		+29.6	*
ms-tpm-20		+28.8	***		+18.0	***
woff2		+22.1	***		+1.6	**
libpcap		+7.3	**		+12.4	***
openssl		+16.5	***		+6.1	**
vorbis		+1.0			-0.9	***
libpng		+4.4			+1.4	**
jsoncpp		+0.0			+0.0	
openh264		+0.3	*		+0.1	
openthread		-2.8			-0.6	
harfbuzz		-2.8	*		+1.1	
libxml2		-2.2			-11.9	***
qpdf		-1.9			-10.3	**
freetype2		-12.7	**		-5.3	**

Table 4: FrameShift coverage compared to AFL++. FrameShift runs are shown as blue bars on top of the centerline; AFL++, red bars beneath it. The left side of a scale represents the lowest coverage obtained by any run, and the right side the most, scaled linearly. Δ% shows average coverage change with FrameShift; p refers to the Mann-Whitney U test p-value: *: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$.

		Reached			Triggered		
Format	**Project**	-	+	p	-	+	p
Binary	libpng	6.0	6.0		3.0	3.0	
	libsndfile	8.0	8.0		7.0	7.0	
	libtiff	12.3	16.2	***	7.9	10.2	**
	openssl	21.0	21.0		3.4	4.4	**
Text	libxml2	15.8	15.2	*	7.3	7.1	
	php	5.0	5.0		2.9	3.0	
	sqlite3	14.5	15.2		5.0	5.2	

Table 5: Average number of bugs *reached* and *triggered* in Magma over 24 hours (10 runs) when FrameShift is disabled (-) and enabled (+) in AFL++. p: Mann-Whitney U test p-values: *: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$

Project / Harness	Bug	HR	p
libpng / libpng_read_fuzzer	PNG007	5.31	**
libsndfile / sndfile_fuzzer	SND005	4.60	*
libsndfile / sndfile_fuzzer	SND017	0.20	**
libtiff / tiff_read_rgba_fuzzer	TIF007	22.35	***
libtiff / tiff_read_rgba_fuzzer	TIF014	0.26	*
libtiff / tiffcp	TIF005	6.26	**
libtiff / tiffcp	TIF006	7.55	**
libtiff / tiffcp	TIF007	32.45	**
libtiff / tiffcp	TIF009	12.10	**
libtiff / tiffcp	TIF012	3.58	*
openssl / client	SSL002	3.18	*
libxml2 / read_memory_fuzzer	XML001	0.04	**
libxml2 / read_memory_fuzzer	XML003	0.03	**
libxml2 / read_memory_fuzzer	XML009	0.11	**
libxml2 / xmllint	XML001	9.99	**
libxml2 / xmllint	XML009	0.21	**

Table 6: Hazard ratios for individual bugs in Magma found by both FrameShift and baseline AFL++. p: Wald test p-values: *: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$

serialized size and/or offset fields that FrameShift identifies, allowing the baseline fuzzer to find differently-sized variants, and thus achieve more coverage, more quickly. Generally, there are two cases: (1) FrameShift enables rapid discovery of core coverage: all of the FrameShift runs end up near the highest found coverage, while baseline fuzzer results are more distributed (for example, `woff2`/AFL++/Empty). Or (2) FrameShift enables breakout coverage discovery, due to unlocking a certain codepath (e.g. `bloaty`/AFL++/Seeded).

Of the neutral benchmarks, several include serialized size/offset fields, yet obtain minimal coverage changes. In `libpng` and `openthread` for example, FrameShift appears to be useful in the first few hours of fuzzing, but the baselines catch up.

We find an interesting case of *frameshift-resistant* file formats, where FrameShift is not able to identify any useful fields. In both `vorbis` and `openh264`, the expected file format contains *sync markers*, explicitly intended to prevent frameshift issues when the file is streamed across an unreliable medium. Thus, the parsers recover when data is improperly resized. As a result, FrameShift does not directly observe frameshifts in the first part of the double-mutant experiment, and recovers no relation fields.

The case most adversarial to FrameShift is when the target generates an extremely large corpus and/or large files. In `harfbuzz`, `libxml2`, `qpdf`, and `freetype2`, the generated corpora are an order of magnitude larger than other benchmarks (tens of thousands of files). Even though FrameShift does identify relations in `harfbuzz` and `freetype2`, it is burdened by the analysis overhead. For text formats, the inability to find relation fields does not directly reduce performance (as demonstrated by `jsoncpp`), but the analysis overhead can reduce the time available to fuzz (as in `libxml2`).

4.4.2 Takeaways. Practitioners should expect to benefit from enabling FrameShift for most binary formats, especially when a seed corpus is unavailable. Incompatible formats incur no direct penalty unless the corpus grows too quickly; for longer campaigns, overhead diminishes as the corpus saturates.

4.5 RQ4: Structure Recovery

In this section we present case studies of FrameShift's ability to identify relations in real-world formats.

4.5.1 ASN.1. Figure 5 (top) shows the relation fields FrameShift finds in a DER-encoded ASN.1 file when run against the `asn1` tool

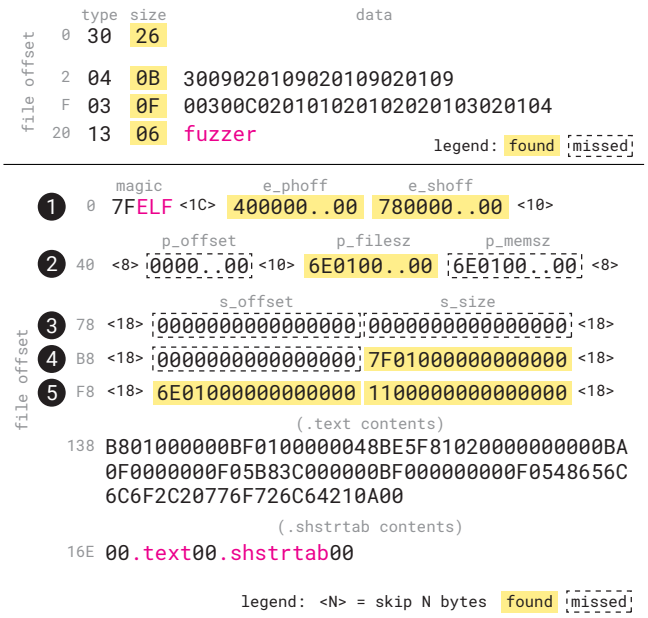


Figure 5: Binary format relation fields FrameShift discovers. Top: DER-encoded ASN.1 file using `openssl`. Bottom: ELF file using `bloaty`.

in `openssl`. This input took 59 milliseconds to analyze and required 63 invocations of the target. The file represents a nested **`SEQUENCE`** object containing three entries: an octet string with values **`[9,9,9]`**, a bitstring with values **`[1,2,3,4]`**, and a printable string with the value **`"fuzzer"`**. Objects are encoded in a TLV (type-length-value) format: a single byte type, a multi-byte length, and an N-sized value.

FrameShift identifies the outer sequence length (at **`0x01`**) and all three of the inner object lengths (at **`0x03`**, **`0x10`**, and **`0x21`**). Length fields in this format are variable-size: values less than 127 are encoded in short form (one byte), while larger values use a prefix byte, **`0x80 + x`**, with x indicating additional length encoding bytes.

Although FrameShift does not explicitly support this variable-size length construction, it still approximates the relation field, because it *appears* to be a single-byte value until the size boundary is crossed. For larger cases, FrameShift ignores the prefix byte and identifies the rest of the field as a relation, which is accurate in most cases as long as the mutation doesn't cause the field to change size. In practice, even though FrameShift is imprecise, it can still be useful, enabling a statistically significant increase in coverage on `openssl` (more than 6%).

4.5.2 ELF. Figure 5 (bottom) visualizes the relation fields identified in an ELF file with the `bloaty` target, which parses several types of object files. Some irrelevant data is omitted for readability, indicated by the <> tags. This file took 458 milliseconds to analyze; FrameShift invoked the target 1571 times during the search. The file consists of an ELF header (**`0-0x40`**), a program table header with one entry (**`0x40-0x78`**), a section table header with three entries (**`0x78-0x138`**), and data mapped into the **`.text`** (**`0x138-0x16E`**) and **`.shstrtab`** (**`0x16E-0x17F`**) sections.

FrameShift correctly identifies the offset fields ❶ for both the program (**`e_phoff`**) and section header tables (**`e_shoff`**). It must discover the latter *first* because insertions before it require shifting both. The program header table contains a single entry (❷), with fields representing on-disk size (**`p_filesz`**) and in memory (**`p_memsz`**) of a segment to map, along with the file offset (**`p_offset`**). FrameShift identifies the file size but not the offset (because the ELF header must start at the beginning) or in-memory size, because `bloaty` only parses, but does not execute, files.

The section header table contains three entries (❸, ❹, and ❺). Each contains, among other metadata, **`s_offset`** and **`s_size`**, describing the offset and size of the section data. The first section in an ELF (❸) is necessarily a **`null`** section, and its size/offset fields are unused, thus FrameShift does not label them as relations. Section (❹) is the **`.text`** section, which maps the entire file. While FrameShift identified **`s_size`** as a relation, it did not label **`s_offset`**, since placing the ELF header anywhere other than the start causes corruption. Finally, ❺ describes the **`.shstrtab`** section which points to a list of null-terminated section names. FrameShift identifies both relations, and can mutate both the data size and position.

4.5.3 Bonus: Automatic Rebase. *Resizing* parts of an ELF file is difficult, given the metadata involved, and typically requires dedicated tools like `lief` [40]. However, while analyzing this ELF example, we were pleasantly surprised to find that FrameShift discovered enough information (in less than half a second) to accurately adjust the metadata while splicing the code region (**`0x138-0x16E`**). In this case, it required fixing three entries: **`p_filesz`** (❷), **`s_size`** for section ❹, and **`s_offset`** for section ❺. The result correctly ran as an executable without additional modification! This small example demonstrates how FrameShift can identify important relations that enable complex, size-changing mutations. This likely contributes to its effectiveness on `bloaty`.

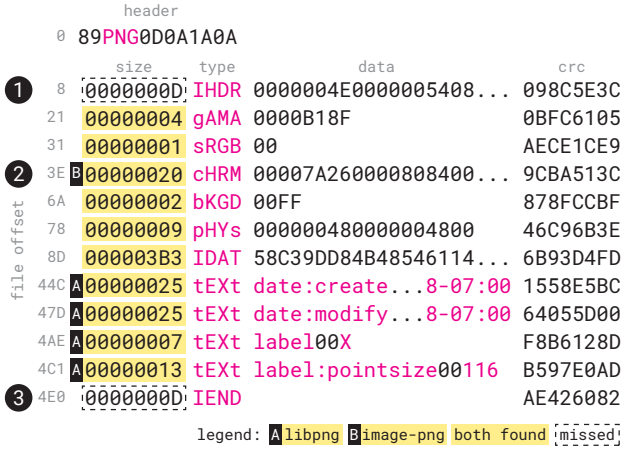
4.6 RQ5: Versatility

FrameShift makes few assumptions about the underlying framework, harness structure, or type of coverage feedback (unlike prior systems that depend on specific toolchains [10]), and it modularly extends baseline fuzzer capabilities, like LibAFL's support for non-C/C++ targets. We demonstrate FrameShift's versatility by applying the LibAFL prototype to two case studies in other languages, Rust (Section 4.6.1) and Python (Section 4.6.2), fuzzing targets that parse similar file formats to at least one C/C++ benchmark program, as well as demonstrating its ability handle ad-hoc binary formats on a SOTA LLM-generated fuzz harness (Section 4.6.3).

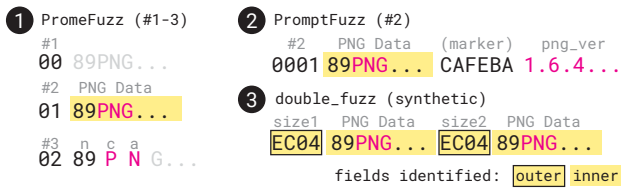
4.6.1 FrameShift in Rust: PNG. Our first target was `image-png`,[10] a Rust library for image encoding and decoding that includes several `cargo-fuzz` harnesses. We compile the target with sanitizer coverage (as we do for C/C++), since the Rust build system also uses LLVM.

We ran the same PNG file through both `image-png` and `libpng` (a C library) for direct comparison. FrameShift took only 313 milliseconds to analyze the file running in `libpng`, invoking the target 4705 times; for `image-png`, it required 1.7 seconds and 9933 target invocations. A static analysis-based tool, would likely incur

[10] https://github.com/image-rs/image-png



(a) PNG file using `libpng` (C) and `image-png` (Rust)



(b) Ad-hoc binary formats

Figure 6: Relations FrameShift identifies in various formats.

a larger overhead switching from C to Rust as Rust binaries contain more boilerplate/bloat. Since FrameShift is dynamic, overhead comes only from runtime performance, which is often tractable.

Figure 6a shows the relation fields identified by the tools. Interestingly, FrameShift discovered *actual* semantic differences between how the two implementations interpret PNG files. A PNG file consists of an 8-byte header followed by variable size chunks. Each chunk has a 4-byte size, 4-byte type, N-byte data, and a 4-byte crc checksum. Nominally, all chunks in a PNG file are resizable. Yet in practice, programs expect (and require!) chunks to be specific sizes. When fuzzing `libpng`, FrameShift only identified 9 out of the possible 12 size fields as relations, ignoring those for the **`IHDR`** (❶), **`cHRM`** (❷), and **`IEND`** (❸) chunks. Analyzing the code, we found that `libpng` does indeed abort if these sizes differ from a set value, and thus attempts to resize them necessarily loses coverage (even when resizing corresponding data).

FrameShift running with `image-png` identifies different relation fields. It *does* label **`cHRM`** as resizable, but none of the **`tEXt`** chunks. Analysis reveals that `image-png` parses the PNG file differently, not immediately validating the size for **`cHRM`**. Furthermore, the harness for `image-png` had a significant difference from `libpng`. The latter iterates through the file to read all chunks; the former is invoked such that it only needs to find the image data, in the **`IDAT`** chunks. This example file contains just one **`IDAT`** chunks, thus `image-png` stops parsing, ignoring the remaining chunks entirely.

From a PNG grammar perspective, these comment chunks have size fields. In the Rust harness, they do not contribute to coverage, thus FrameShift (rightly) ignores them. This example demonstrate that FrameShift not only transparently works across languages, but can reveal implementation-specific behaviors, suggesting value for understanding semantic differences between programs that nominally handle the same format.

4.6.2 FrameShift in Python: ASN.1. We also evaluated FrameShift on a Python case study, which uses a different compiler toolchain and alternative coverage feedback. Such an adaptation would be fundamentally difficult for a static analysis-based approach, while FrameShift supports it out-of-the-box given LibAFL's support for Atheris [17], a coverage-guided Python fuzzer. Python represents a higher abstraction level than C/C++, e.g., list concatenation in a C++ program may hit many basic blocks, but is a single opcode in Python. This changes the nature of coverage instrumentation.

Our target program was `pyasn1`,[11] a Python-based framework for encoding and decoding ASN.1 files. We used the same ASN.1 file from Figure 5 (top) as a seed. Running the analysis took 25 milliseconds and 125 invocations of the target program, and found the same fields as it did on the `openssl` benchmark. In this case, however, it also incorrectly identified the byte at position **`0x20`** as a size field. This is because when FrameShift corrupted this field and performed a nearby insertion, it coincidentally achieved enough of the same coverage features (through some other mechanism) to suggest it is a relation. Increasing the t_{loss} threshold was sufficient to remove this false positive. This suggests that FrameShift's parameters may benefit from tuning when applying it to frameworks with different types of coverage feedback.

4.6.3 Ad-hoc Binary Formats. Libary API harnesses often split the fuzzer-provided byte sequence into pieces to decide which API functions to invoke with what data. For example, a random byte stream may be interpreted as custom configuration followed by the documented format. As a result, the harness's nominal input format (e.g., PNG) is often embedded as part of a larger *ad-hoc* binary format. Such formats are likely to become increasingly prevalent, given the advance of LLM-generated harnesses [28, 29].

FrameShift naturally handles ad hoc formats. Figure 6b shows three `libpng` harnesses that perform additional parsing and branching beyond the png grammar. PromeFuzz [28] (❶) and PromptFuzz [29] (❷) use LLMs to automatically generate harnesses [12], integrated with large switch statements, while our synthetic demonstration harness (❸) stress-tests FrameShift by invoking the OSS-Fuzz harness for `libpng` twice with size-prefixed data chunks.

In PromeFuzz (❶), the first byte selects which sub-harness runs. Of the first three sub-harnesses, only the second interprets the data as PNG (passing it to `libpng` parsers); the others either ignore the data entirely or parse only the first three bytes. In PromptFuzz (❷), two bytes select the sub-harness, only some of which actually interpret data as PNG. For example, harness #2, splits the data stream using `FuzzedDataProvider` using the sequence `CAFEBA` as a magic marker to delimit the PNG data.

While these harnesses would stump traditional format-guided fuzzers (which produce raw PNG data, not PNG with this extra metadata), FrameShift identifies the same relation fields as in Figure 6a, correctly placed within the input stream. FrameShift also handles

[11] https://github.com/etingof/pyasn1
[12] 127 and 117 harnesses, respectively, for `libpng`

our synthetic harness that parses two size-prefixed data chunks and passes them to `libpng` (❸). It identifies both the outer size-prefix fields wrapping the PNG data, and the same inner PNG chunk size fields for *both* chunks.

5 Discussion

Future Directions. FrameShift is an example of a potentially more general class of techniques which use coverage feedback with heuristics to search input structures and augment fuzzers. This need not be limited just to size and offset fields. For example, a similar double-mutant experiment may be able to identify compressed/encoded input regions (e.g. `zlib`, `base64`, etc.) or repeatable input parts (i.e. chunks). A key challenge, as we have seen first-hand in prototyping these ideas, is keeping the analysis time sufficiently low to produce wins in the resulting structure-aware fuzzing.

Another interesting future direction could augment this type of coverage-guided inference with a more powerful heuristic, such as a large language model, to propose high-quality structure candidates. It is possible that such a technique could discover much more complex (and thus more useful) structures, that could then be validated dynamically; the additional complexity might effectively counterbalance additional analysis time.

Finally, it may be possible to automatically tune FrameShift parameters based on the target program, such as t_{loss} (as discussed for Python), or to limit analysis on a fast-growing corpus.

Structure Inference in the LLM Era. Recent LLM advancements for structure-aware fuzzing primarily *generate* high-quality valid inputs, either directly [43], or by constructing generator functions [46]. Like high-quality seed corpora, these techniques unlock broad code coverage. However, LLMs alone have not replaced traditional mutation engines. For example, G^{2}FUZZ [46] augments AFL++ with an LLM component to synthesize input generators; their ablation found that removing the underlying fuzzer (leaving only the LLM) significantly reduced effectiveness.

FrameShift complements such tools by helping the mutation engine *mutate* inputs without destroying important structure (specifically, relation fields). Importantly, FrameShift is grounded by actual program execution, not hypothetical format specifications. As shown in Section 4.6, ground truth structures are *program- and input-specific*; even targets parsing the same nominal format may enforce different requirements. LLMs may eventually play a role in this sort of coverage-guided structure-inference, as discussed.

LLMs are also promising for generating fuzz harnesses, increasing the number of API methods that can be automatically targeted in various configurations. However, such harnesses create new ad hoc binary formats that stymie grammar- or format-guided fuzzers; Section 4.6 shows that FrameShift is effective in these applications, making structure inference even more valuable.

6 Related Work

Structure inference for binary fuzzing. Tools most similar to FrameShift perform automatic structure-inference to aid fuzzers. NestFuzz [10] uses dynamic taint analysis (DTA) to learn input processing logic; TIFF [22] uses DTA to infer input field types. AIFORE [36] fuses byte-level taint analysis with machine learning for clustering.

WEIZZ [12] and ProFuzzer [44] both take a greybox approach, like FrameShift, using coverage feedback identify structure. The key novelty of FrameShift is to identify relation fields and associated spans, including nested fields, enabling correct simultaneous field/buffer resize updates. WEIZZ can identify size/offset bytes but doesn't distinguish them from other field types, identify associated data spans, or perform valid resizing. ProFuzzer heuristically identifies size/offset fields based on how mutations affect coverage (similar to FrameShift's first double-mutation step), but when incrementing an offset/size field, only tries inserting one byte after the field or at file end. It cannot perform reciprocal mutations where resizing input affects size/offsets, nor identify nested fields. Finally, FrameShift interoperates with existing byte-mutators, leveraging innovations like compare-logging; WEIZZ and ProFuzzer reimplement the mutation layer.

Structure inference for reverse-engineering. A parallel body of work recovers *internal* structures to aid decompilation or static analysis [8, 9, 24–26, 37]. While conceptually similar, these approaches target the structures used internally to the program, not the serialized input structure. Thus, they are not immediately useful for fuzzing. We show an instance of FrameShift identifying interesting differences between programs parsing similar input formats, suggesting potential to help humans understand program behavior.

Specification-based fuzzing. An alternative approach provides a fuzzer with a specification beforehand. AFLSmart [32] performs smart chunk-based mutations when provided the virtual structure of a file format which are similar to the types of mutations FrameShift enables, but require manual specification. FormatFuzzer [11] repurposes structure format files used by a file structure explorer utility, converting them into data *generators* and *mutators*. Similarly, the ISLa [39] project aims to create an input specification language that can be sampled using a constraint solver. These approaches are interesting and useful when such formats are available. However, they can be onerous to provide, motivating techniques like FrameShift. Programs may also parse multiple formats at once (e.g., `bloaty`), or implement a format's semantics *differently* than other programs. FrameShift can learn program-specific formats (as it did with PNG); specification-driven fuzzers cannot.

Grammar-based fuzzing. Structure-aware fuzzing has been heavily utilized for text-based formats like scripting language interpreters [1, 16, 19, 20, 33, 41, 47]. These formats are usually representable with a context-free grammar (CFG) and do not contain serialized size/offset fields, avoiding the *frameshift* problem. Tools like GLADE [4], Pygmalion [18], Skyfire [42], and Autogram [21] automatically learn such grammars. Given existing grammars, Nautilus [1] and Gramatron [38] can perform coverage-guided semantic-preserving mutations. Perhaps the most related work in this domain is Grimoire [5], which upon receiving a new input tries to understand how to *generalize* it by observing how different mutations change coverage.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by the National Science Foundation, CNS-2120696.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *NDSS*.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[3] Cornelius (eqv / .eqv) Aschermann. 2024. "something that I've seen commonly that made fuzzing VERY difficult is TLV kind of formats, where the header has an overall size field , and each chunk has a size field, and we have sizeof(header)+header.size + sum(chunks.size) == file_size and sum(chunks.size) == header.size" — Discord message in *#general* (Awesome Fuzzing server). https://discord.com/channels/736989425770168422/736989426298912813/1287160766469767199 Message ID 1287160766469767199; posted 2024-09-21 17:17; accessed 2025-06-05.

[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110.

[5] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002.

[6] 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.

[7] David R Cox. 1972. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)* 34, 2 (1972), 187–202.

[8] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces.. In *USENIX Security Symposium*. Boston, MA, USA, 1–14.

[9] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*. 391–402.

[10] Peng Deng, Zhemin Yang, Lei Zhang, Guangliang Yang, Wenzheng Hong, Yuan Zhang, and Min Yang. 2023. NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1272–1286.

[11] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2023. Formatfuzzer: Effective fuzzing of binary file formats. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–29.

[12] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 1–13.

[13] 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)*.

[14] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)* (Los Angeles, U.S.A.) *(CCS '22)*. ACM.

[15] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. 2025. LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 355–373.

[16] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

[17] Google. 2023. Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris. Version 2.3.0, commit cbf4ad9, accessed 2025-06-06.

[18] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. 2018. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289* (2018).

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[21] Matthias Hoschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 31–34.

[22] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: using input type inference to improve fuzzing. In *Proceedings of the 34th annual computer security applications conference*. 505–517.

[23] 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.

[24] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[25] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 83–93.

[26] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. 1–1.

[27] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. 2023. SBFT tool competition 2023-Fuzzing track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 51–54.

[28] Yuwei Liu, Junquan Deng, Xiangkun Jia, Yanhao Wang, Minghua Wang, Lin Huang, Tao Wei, and Purui Su. 2025. PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1559–1573.

[29] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3793–3807.

[30] 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*. 1393–1403.

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

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

[33] Jesse Ruderman. 2007. Introducing jsfunfuzz. *URL http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz* 20 (2007), 25–29.

[34] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.

[35] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[36] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. 2023. {AIFORE}: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4967–4984.

[37] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures.. In *NDSS*.

[38] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.

[39] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 583–594.

[40] Romain Thomas. 2017. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/.

[41] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[43] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[44] 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.

[45] Michal Zalewski. [n. d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: September 10, 2024.

[46] 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).

[47] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards better semantics exploration for browser fuzzing. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 604–631.