# WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats

Andrea Fioraldi
Sapienza University of Rome
Italy
andreafioraldi@gmail.com

Daniele Cono D'Elia
Sapienza University of Rome
Italy
delia@diag.uniroma1.it

Emilio Coppa
Sapienza University of Rome
Italy
coppa@diag.uniroma1.it

## ABSTRACT

Fuzzing technologies have evolved at a fast pace in recent years, revealing bugs in programs with ever increasing depth and speed. Applications working with complex formats are however more difficult to take on, as inputs need to meet certain format-specific characteristics to get through the initial parsing stage and reach deeper behaviors of the program.

Unlike prior proposals based on manually written format specifications, we propose a technique to automatically generate and mutate inputs for unknown chunk-based binary formats. We identify dependencies between input bytes and comparison instructions, and use them to assign tags that characterize the processing logic of the program. Tags become the building block for structure-aware mutations involving chunks and fields of the input.

Our technique can perform comparably to structure-aware fuzzing proposals that require human assistance. Our prototype implementation Weizz revealed 16 unknown bugs in widely used programs.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software verification and validation*; • **Security and privacy** → Software and application security.

## KEYWORDS

Fuzzing, binary testing, chunk-based formats, structural mutations

**ACM Reference Format:**
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 (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397372

## 1 INTRODUCTION

Recent years have witnesses a spike of activity in the development of efficient techniques for fuzz testing, also known as fuzzing. In particular, the coverage-based grey-box fuzzing (CGF) approach has proven to be very effective for finding bugs often indicative of weaknesses from a security standpoint. The availability of the AFL fuzzing framework [36] paved the way to a large body of works proposing not only more efficient implementations, but also new techniques to deal with common fuzzing roadblocks represented by magic numbers and checksum tests in programs. Nonetheless, there are still several popular application scenarios that make hard cases even for the most advanced CGF fuzzers.

CGF fuzzers operate by mutating inputs at the granularity of their bit and byte representations, deeming mutated inputs interesting when they lead execution to explore new program portions. While this approach works very well for compact and unstructured inputs [30], it can lose efficacy for highly structured inputs that must conform to some grammar or other type of specification.

Intuitively, "undisciplined" mutations make a fuzzer spend important time in generating many inputs that the initial parsing stages of some program typically rejects, resulting in little-to-none code coverage improvement. Researchers thus have added a user-supplied specification to the picture to produce and prioritize meaningful inputs: enhanced CGF embodiments of this kind are available for both grammar-based [4, 30] and chunk-based [24] formats, with the latter seeming prevalent among real-world software [24].

The shortcomings of this approach are readily apparent. Applications typically do not come with format specifications suitable to this end. Asking users to write one is at odds with a driving factor behind the success of CGF fuzzers, that is, they work with a minimum amount of prior knowledge [5]. Such a request can be expensive, and hardly applies to security contexts where users deal with proprietary or undocumented formats [8]. The second limitation is that testing only inputs perfectly adhering to the specification would miss imprecisions in the implementation, while inputs that are to some degree outside it may instead exercise them [8].

**Our approach.** The main feature of our proposal can be summarized as: *we attempt to learn how some chunk-based input structure may look like based on how a program handles its bytes.*

Nuances of this idea are present in [8] where code coverage guides grammar structure inference, and to some extent in a few general-purpose fuzzing additions. For instance, taint tracking can reveal which input bytes take part in comparisons with magic sequences [10, 25], while input-to-state relationships [5] can identify also checksum fields using values observed as comparison operands.

We build on the intuition that among comparisons operating on input-derived data, we can deem some as tightly coupled to a single portion of the input structure. We also experimentally observed that the order in which a program exercises these checks can reveal structural details such as the location for the type specifier of a chunk. We tag input bytes with the most representative comparison for their processing, and heuristically infer plausible boundaries for chunks and fields. With this automatic pipeline, we can apply structure-aware mutations for chunk-based grey-box fuzzing [24] without the need for a user-supplied format specification.

We start by determining candidate instructions that we can consider relevant with respect to an input byte. Instead of resorting to taint tracking, we flip bits in every input byte, execute the program, and build a dependency vector for operands at comparison instruction sites. We analyze dependencies to identify input-to-state relationships and roadblocks, and to assign tags to input bytes. Tag assignment builds on spatial and temporal properties, as a program typically uses distinct instructions in the form of comparisons to parse distinct items. To break ties we prioritize older instructions as format validation normally happens early in the execution. Tags drive the inference process of an approximate chunk-based structure of the input, enabling subsequent structure-aware mutations.

Unlike reverse engineering scenarios for reconstructing specifications of input formats [12, 18, 19], experimental results suggest that in this context an inference process does not have to be fully accurate: fuzzing can deal with some amount of noise and imprecision. Another important consideration is that a specification does not prescribe how its implementation should look like. Developers can share code among functionalities, introducing subtle bugs: we have observed this phenomenon for instance in applications that operate on multiple input formats. Also, they can devise optimized variants of one functionality, specialized on e.g. value ranges of some relevant input field. As we assign tags and attempt inference over one input instance at a time, we have a chance to identify and exercise such patterns when aiming for code coverage improvements.

**Contributions.** By using information that is within immediate reach of a fuzzer in its usual work, we propose a method to infer an approximate structure for a chunk-based input and then mutate it in a fully automatic manner. Throughout the paper we present:

- a dependency identification technique embedded in the deterministic mutation stage of grey-box fuzzing;
- a tag assignment mechanism that uses such dependencies to overcome fuzzing roadblocks and to back a structure inference scheme for chunk-based formats;
- an implementation of the approach called WEIZZ.

In our experiments WEIZZ beats or matches a chunk-based CGF proposal that requires a format specification, and outperforms several general-purpose fuzzers over the applications we considered. We make WEIZZ available as open source at:

https://github.com/andreafioraldi/weizz-fuzzer

## 2 STATE OF THE ART

The last years have seen a large body of fuzzing-related works [37]. Grey-box fuzzers have gradually replaced initial *black-box* fuzzing proposals where mutation decisions happen without taking into account their impact on the execution path [29]. Coverage-based *grey-box* fuzzing (CGF) uses lightweight instrumentation to measure code coverage, which discriminates whether mutated inputs are sufficiently "interesting" by looking at control-flow changes.

CGF is very effective in discovering bugs in real software [24], but may struggle in the presence of roadblocks (like magic numbers and checksums) [5], or when mutating structured grammar-based [8] or chunk-based [24] input formats. In this section we will describe the workings of the popular CGF engine that WEIZZ builds upon, and recent proposals that cope with the challenges listed above. We conclude by discussing where WEIZZ fits in this landscape.

### 2.1 Coverage-Based Fuzzing

American Fuzzy Lop (AFL) [36] is a very well-known CGF fuzzer and several research works have built on it to improve its effectiveness. To build new inputs, AFL draws from a queue of initial user-supplied *seeds* and previously generated inputs, and runs two mutation stages. Both the *deterministic* and the *havoc* nondeterministic stage look for coverage improvements and crashes. AFL adds to the queue inputs that improve the coverage of the program, and reports crashing inputs to users as proofs for bugs.

**Coverage.** AFL adds instrumentation to a binary program to intercept when a branch is hit during the execution. To efficiently track *hit counts*, it uses a coverage map of branches indexed by a hash of its source and destination basic block addresses. AFL deems an input *interesting* when the execution path reaches a branch that yields a previously unseen hit count. To limit the number of monitored inputs and discard likely similar executions, hit counts undergo a normalization step (power-of-two buckets) for lookup.

**Mutations.** The deterministic stage of AFL sequentially scans the input, applying for each position a set of mutations such as bit or byte flipping, arithmetic increments and decrements, substitution with common constants (e.g., 0, -1, `MAX_INT`) or values from a user-supplied *dictionary*. AFL tests each mutation in isolation by executing the program on the derived input and inspecting the coverage, then it reverts the change and moves to another.

The havoc stage of AFL applies a nondeterministic sequence (or *stack*) of mutations before attempting execution. Mutations come in a random number (1 to 256) and consists in flips, increments, decrements, deletions, etc. at a random position in the input.

### 2.2 Roadblocks

*Roadblocks* are comparison patterns over the input that are intrinsically hard to overcome through blind mutations: the two most common embodiments are *magic numbers*, often found for instance in header fields, and *checksums*, typically used to verify data integrity. Format-specific dictionaries may help with magic numbers, yet the fuzzer has to figure out where to place such values. Researchers over the years have come up with a few approaches to handle magic numbers and checksums in grey-box fuzzers.

**Sub-instruction Profiling.** While understanding how a large amount of logic can be encoded in a single comparison is not trivial, one could break multi-byte comparisons into single-byte [1] (or even single-bit [22]) checks to better track progress when attempting to match constant values. LAF-INTEL [1] and COMPARECOVERAGE [20] are compiler extensions to produce binaries for such a fuzzing. HONGGFUZZ [28] implements this technique for fuzzing programs when the source is available, while AFL++ [16] can automatically transform binaries during fuzzing. STEELIX [21] resorts to static analysis to filter out likely uninteresting comparisons from profiling. While sub-instruction profiling is valuable to overcome tests for magic numbers, it is ineffective however with checksums.

**Taint Analysis.** Dynamic taint analysis (DTA) [26] tracks when and which input parts affect program instructions. VUZZER [25] uses DTA for checks on magic numbers in binary code, identified as comparison instructions where an operand is input-dependent and the other is constant: VUZZER places the constant value in the input portion that propagates directly to the former operand. ANGORA [10] brings two improvements: it identifies magic numbers that are not contiguous in the input with a multi-byte form of DTA, and uses gradient descent to mutate tainted input bytes efficiently. It however requires compiler transformations to enable such mutations.

**Symbolic Execution.** DTA techniques can at best identify checksum tests, but not provide sufficient information to address them. Several proposals (e.g., [23, 27, 31, 35]) use symbolic execution [6] to identify and try to solve complex input constraints involved in generic roadblocks. TAINTSCOPE [31] identifies possible checksums with DTA, patches them away for the sake of fuzzing, and later tries to repair inputs with symbolic execution. It requires the user to provide specific seeds and is subject to false positives [23, 27].

T-FUZZ [23] makes a step further by removing the need for specific seeds and extends the set of disabled checks during fuzzing to any sanity (*non-critical*) check that is hard to bypass for a fuzzer but not essential for the computation. For instance, magic numbers can be safely ignored to help fuzzing, while a check on the length of a field should not be patched away. As in TAINTSCOPE, crashing inputs are repaired using symbolic execution or manual analysis.

DRILLER [27] instead uses symbolic execution as an alternate strategy to explore inputs, switching to it when fuzzing exhausts a budget obtaining no coverage improvement. Similarly, QSYM [35] builds on concolic execution implemented via dynamic binary instrumentation [13] to trade exhaustiveness for speed.

**Approximate Analyses.** A recent trend is to explore solutions that approximately extract the information that DTA or symbolic execution can bring, but faster.

REDQUEEN [5] builds on the observation that often input bytes flow directly, or after simple encodings (e.g. swaps for endianness), into instruction operands. This **input-to-state** (I2S) correspondence can be used instead of DTA and symbolic execution to deal with magic bytes, multi-byte compares, and checksums. REDQUEEN approximates DTA by changing input bytes with random values (*colorization*) to increase the entropy in the input and then looking for matching patterns between comparisons operands and input parts, which would suggest a dependency. When both operands of a comparison change, but only for one there is an I2S relationship, REDQUEEN deems it as a likely checksum test. It then patches the operation, and later attempts to repair the input with educated guesses consisting in mutations of the observed comparison operand values. A topological sort of patches addresses nested checksums.

SLF [34] attemps a dependency analysis to deal with the generation of valid input seeds when no meaningful test cases are available. It starts from a small random input and flips individual bits in each byte, executing the program to collect operand values involved in comparisons. When consecutive input bytes affect the same set of comparisons across the mutations, they are marked as part of the same field. SLF then uses heuristics on observed values to classify checks according to their relations with the exercised inputs: its focus are identifying offsets, counts, and length of input fields, as they can be difficult to reason about for symbolic executors.

In the context of concolic fuzzers, ECLIPSER [11] relaxes the path constraints over each input byte. It runs a program multiple times

**Table 1: Comparison with related approaches.**

| Fuzzer | Binary | Magic bytes | Checksums | Chunk-based | Grammar-based | Automatic |
|---|---|---|---|---|---|---|
| AFL++ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ |
| ANGORA | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| ECLIPSER | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ |
| REDQUEEN | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| STEELIX | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ |
| NAUTILUS | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ |
| SUPERION | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ |
| GRIMOIRE | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| AFLSMART | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| WEIZZ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |

by mutating individual input bytes to identify the affected branches. Then it collects constraints resulting from them, considering however only linear and monotone relationships, as other constraint types would likely require a full-fledged SMT solver. ECLIPSER then selects one of the branches and flips its constraints to generate a new input, mimicking dynamic symbolic execution [6].

## 2.3 Format-Aware Fuzzing

Classic CGF techniques lose part of their efficacy when dealing with structured input formats found in files. As mutations happen on bit-level representations of inputs, they can hardly bring the structural changes required to explore new compartments of the data processing logic of an application. Format awareness can however boost CGF: in the literature we can distinguish techniques targeting *grammar-based* formats, where inputs comply to a language grammar, and *chunk-based* ones, where inputs follow a tree hierarchy with C structure-like data chunks to form individual nodes.

**Grammar-Based Fuzzing.** LANGFUZZ [17] generates valid inputs for a Javascript interpreter using a grammar, combining in a black-box manner sample code fragments and test cases. NAUTILUS [4] and SUPERION [30] are recent grey-box fuzzer proposals that can test language interpreters without requiring a large corpus of valid inputs or fragments, but only an ANTLR grammar file.

GRIMOIRE [8] then removes the grammar specification requirement. Based on REDQUEEN, it identifies fragments from an initial set of inputs that trigger new coverage, and strips them from parts that would not cause a coverage loss. GRIMOIRE notes information for such gaps, and later attempts to recursively splice-in parts seen in other positions, thus mimicking grammar combinations.

**Chunk-Based Fuzzing.** SPIKE [3] lets users describe the network protocol in use for an application to improve black-box fuzzing. PEACH [15] generalizes this idea, applying format-aware mutations on an initial set of valid inputs using a user-defined input specification dubbed *peach pit*. As they are input-aware, some literature calls such black-box fuzzers *smart* [24]. However a smart grey-box variant may outperform them, as due to the lack of feedback (as in explored code) they do not keep mutating interesting inputs.

AFLSMART [24] validates this speculation by adding smart (or *high-order*) mutations to AFL that add, delete, and splice chunks in an input. Using a peach pit, AFLSMART maintains a *virtual structure* of the current input, represented conceptually by a tree whose internal nodes are chunks and leaves are attributes. Chunks are characterized by initial and end position in the input, a format-specific chunk type, and a list of children attributes and nested chunks. An attribute is a *field* that can be mutated without altering the structure. Chunk addition involves adding as sibling a chunk taken from another input, with both having a parent node of the same type. Chunk deletion trims the corresponding input bytes. Chunk splicing replaces data in a chunk using a chunk of the same type from another input. As virtual structure construction is expensive, AFLSmart defers smart mutations based on the time elapsed since it last found a new path, as trying them over every input would make AFLSmart fall behind classic grey-box fuzzing.

## 2.4 Discussion

Tables 1 depicts where Weizz fits in the state of the art of CGF techniques mentioned in the previous sections. Modern general-purpose CGF fuzzers (for which we choose a representative subset with the first five entries) can handle magic bytes, but only RedQueen proposes an effective solution for generic checksums. In the context of grammar-based CGF fuzzers, Grimoire is currently the only fully automatic (i.e., no format specification needed) solution, and can handle both magic bytes and checksums thanks to the underlying RedQueen infrastructure. With Weizz we bring similar enhancements to the context of chunk-based formats, proposing a scheme that at the same time can handle magic bytes and checksums and eliminates the need for a specification that characterizes AFLSmart.

## 3 METHODOLOGY

The fuzzing logic of Weizz comprises two stages, depicted in Figure 1. Both stages pick from a shared queue made of inputs processed by previous iterations of either stage. The surgical stage identifies dependencies between an input and the comparisons made in the program: it summarizes them by placing *tags* on input bytes, and applies deterministic mutations to the sole bytes that turn out to influence operands of comparison instructions. The structure-aware stage extends the nondeterministic working of AFL, leveraging previously assigned tags to infer the location of fields and chunks in an input and mutate them. In the next sections we will detail the inner workings of the two stages.

## 3.1 Surgical Stage

The surgical stage replaces the deterministic working of AFL, capturing in the process dependency information for comparison instructions that is within immediate reach of a fuzzer. Weizz summarizes it for the next stage by placing tags on input bytes, and uses it also to identify I2S relationships (Section 2.2) and checksums.

The analysis is context-sensitive, that is, when analyzing a comparison we take into account its calling context [14]: the *site* of the comparison is computed as the exclusive OR of the instruction address with the word used to encode the calling context.

Weizz also maintains a global structure $CI$ to keep track of comparison instructions that the analysis of one or more inputs indicated as possibly involved in checksum tests.

We will use the running example of Figure 2 to complement the description of each component of the surgical stage. Before detailing them individually, we provide the reader with an overview of the workflow that characterizes this stage as an input enters it.

*3.1.1 Overview.* Given an input $I$ picked from the queue, Weizz attempts to determine the dependencies between every bit of $I$ taken individually and the comparison instructions in the program.

Procedure GetDeps builds two data structures, both indexed by a hash function $Sites$ for comparison sites. A comparison table

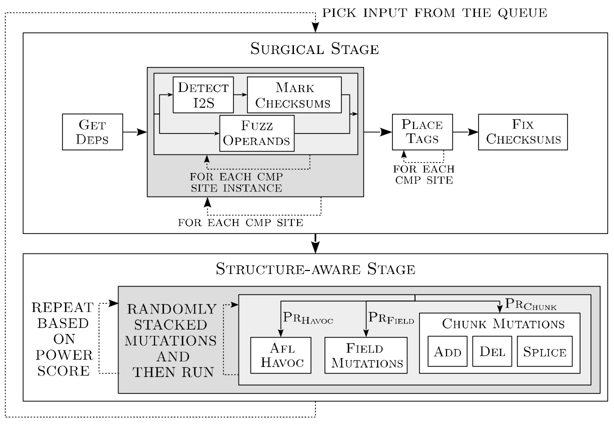


**Figure 1: Two-stage architecture of Weizz.**

$CT$ stores values for operands involved in observed instances of comparison instructions at different sites. For such operands $Deps$ stores which bytes in the input can influence them.

Weizz then moves to analyzing the recorded information. For each site, it iterates over each instance present in $CT$ looking for I2S dependencies (DetectI2S $\mapsto$ R) and checksum information (MarkChecksums $\mapsto$ CI), and mutates bytes that can alter comparison operands leveraging recorded values (FuzzOperands).

The stage then moves to tag construction, with each input byte initially untagged. It sorts comparison sites according to when they were first encountered and processes them in this order. Procedure PlaceTags assigns a tag to each input byte taking into account I2S relationships $R$, checksum information $CI$, the initially computed dependencies $Deps$, and data associated with comparison sites $CT$.

Like other fuzzers [5, 31] Weizz forces checksums by patching involved instructions as it detects one, postponing to the end of the surgical stage the input repairing process FixChecksums required to meet the unaltered checksum condition. To this end we use a technique very similar to the one of RedQueen (Section 2.2).

The output of the surgical stage is an input annotated with discovered tags and repaired to meet checksums, and enters the queue ready for the structure-aware stage to pick it up.

The procedures for which we report the name with underlined style are described only informally in this paper: the reader can find their full pseudocode in our extended online technical report[1].

*3.1.2 Dependency Identification.* A crucial feature that makes a fuzzer effective is the ability to understand how mutations over the input affect the execution of a program. In this paper we go down the avenue of fast analyses to extract approximate dependency information. We propose a technique that captures which input bytes individually affect comparison instructions: it can capture I2S facts like the technique of RedQueen, but also non-I2S relationships that turn out to be equally important to assign tags to input bytes.

Algorithm 1 describes the GetDeps procedure used to this end. Initially we run the program over the current input instrumenting all comparison instructions. The output is a comparison table $CT$ that records the operands for the most recent $|J|$ observations (*instances*) of a comparison site. For each comparison site $CT$ keeps

[1]https://arxiv.org/abs/1911.00621.

```
       void parser(char* input, int len) {
          unsigned short id, size, expected, ck = 0;
          int i, offset = 0;
          id = read_ushort(input + offset);
          offset += sizeof(id);
          size = read_ushort(input + offset);
          offset += sizeof(size);
CMP_A:    if (id >= 0xAAAA) exit(1);
CMP_B:    if (size > len - 3*sizeof(short)) exit(1);
          offset += size;
CMP_C:    for (i = 0; i < offset; ++i)
              ck ^= input[i] << (i % 8);
          expected = read_ushort(input + offset);
CMP_D:    if (ck != expected) exit(1);

          // process id and data
       }
```

**(a)**

**(b)** Input format: `[id] [size] [data] [checksum]`
where: `id`, `size`, and `checksum` are 2-byte long
`data` is `size`-byte long

Bytes of seed:
`[0E 00][02 00][41 41][36 0C]`

Collected comparison instances:

| $CMP_A$ | $CMP_B$ | $CMP_C$ | $CMP_D$ |
|---|---|---|---|
| E, AAAA | 2, 2 | 0, 6 | C36, C36 |
| | | 1, 6 | |
| | | 2, 6 | |
| | | 3, 6 | |
| | | 4, 6 | |
| | | 5, 6 | |
| | | 6, 6 | |

**(c)**

Bytes of seed after flipping first bit:
[**8**E 00][02 00][41 41][36 0C]

Collected comparison instances:

| $CMP_A$ | $CMP_B$ | $CMP_C$ | $CMP_D$ |
|---|---|---|---|
| **8E**, AAAA | 2, 2 | 0, 6 | **CB6**, C36 |
| | | 1, 6 | |
| | | 2, 6 | |
| | | 3, 6 | |
| | | 4, 6 | |
| | | 5, 6 | |
| | | 6, 6 | |

**(d)**

**Figure 2: Example for surgical stage: (a) code of function `parser`; (b) input format; comparison instances collected by WEIZZ when running `parser` on (c) the initial seed and (d) on the seed after flipping its first bit *(affected operands are marked in bold).***

```
CT: Sites × J × {op_1, op_2} → V       [cmp site instance & operand → value]
Deps: Sites × J × {op_1, op_2} → A     [cmp site inst. & operand → array of |input| booleans]
function GetDeps(I):
1  CT ← RunInstr(I)
2  foreach b ∈ {0 ... len(I)-1} do
3      Deps(s, j, op)[b] ← false ∀(s, j, op) ∈ dom(CT)
4      foreach k ∈ {0 ... 7} do
5          CT′ ← RunInstr(BitFlip(I, b, k))
6          foreach (s, j, op) ∈ dom(CT) do
7              if Hits(CT, s) ≠ Hits(CT′, s) then continue
8              Deps(s, j, op)[b] ← Deps(s, j, op)[b] ∨ CT(s, j, op) ≠ CT′(s, j, op)
9  return CT, Deps
```

**Algorithm 1:** Dependency identification step.

a timestamp for when RunInstr first observed it, and how many times it encountered such site in the execution (Hits at line 7).

WEIZZ then attempts to determine which input bytes contribute to operands at comparison sites, either directly or through derived values. By altering bytes individually, WEIZZ looks for sites that see their operands changed after a mutation. The algorithm iterates over the bits of each byte, flipping them one at a time and obtaining a $CT'$ for an instrumented execution with the new input (line 5). We mark the byte as a dependency for an operand of a comparison site instance (line 8) if its $b$-th byte has changed in $CT'$ w.r.t. $CT$.

Bit flips may however cause execution divergences, as some comparisons likely steer the program towards different paths (and when such an input is *interesting* we add it to the queue, see Section 2.1). Before updating dependency information, we check whether a comparison site $s$ witnessed a different number of instances in the two executions (line 7). This is a proxy to determine whether execution did not diverge locally at the comparison site. We prefer a local policy over enforcing that all sites in $CT$ and $CT'$ see the same number of instances. In fact for a mutated input some code portions can see their comparison operands change (suggesting a dependency) without incurring control-flow divergences, while elsewhere the two executions may differ too much to look for correlations.

**Example.** We discuss a simple parsing code (Figure 2a) for a custom input format characterized by three fields `id`, `size`, and `checksum`, each expressed using 2 bytes, and a field `data` of variable length encoded by the `size` field (Figure 2b).

Our parser takes as argument a pointer `input` to the incoming bytes and their number `len`. It assumes the input buffer will contain at least 6 bytes, which is the minimum size for a valid input holding empty data. The code operates as follows:

(a) it reads the `id` and `size` fields from the buffer;
(b) it checks the validity of field `id` (label `CMP_A`);
(c) it checks whether the buffer contains at least 6+`size` bytes to host `data` and the other fields (label `CMP_B`);
(d) it computes a checksum value iterating (label `CMP_C`) over the bytes associated with `id`, `size`, and `data`;
(e) it reads the `expected` checksum from the buffer and validates the one presently computed (label `CMP_D`).

Figure 2c shows the comparison instances collected by WEIZZ when running the function `parser` over the initial seed. For brevity we omit timestamps and hit counts and we assume that the calling context is not relevant, so we can use comparison labels to identify sites. Multiple instances appear for site `CMP_C` as it is executed multiple times within a loop.

Figure 2d shows the comparison instances collected after flipping one bit in the first byte of the seed, with changes highlighted in bold. Through these changes WEIZZ detects that the first operand of both `CMP_A` and `CMP_D` was affected, revealing dependencies between these two comparison sites and the first input byte. Subsequent flips will lead to inputs that reveal further dependencies:

- the 1st operand of `CMP_A` depends on the bytes of `id`;
- the 1st oper. of `CMP_B` depends on the bytes of `size`;
- the 2nd oper. of `CMP_C` depends on the 1st byte of `size`[2];
- the 1st oper. of `CMP_D` depends on the bytes of both `id` and `data`, as well as on the first byte of `size`, while the 2nd oper. is affected by the `checksum` field bytes;

### 3.1.3 *Analysis of Comparison Site Instances.*

After constructing the dependencies, WEIZZ starts processing the data recorded for up to $|J|$ instances of each comparison site. The first analysis is the detec-

[2]The dependency is exposed only when flipping the second least significant bit of `size`, turning it into the value zero, as flipping other bits makes the size invalid and aborts the execution prematurely at `CMP_B`.

tion of I2S correspondences with DetectI2S. RedQueen explores this concept to deal with roadblocks, based on the intuition that parts of the input can flow directly into the program state in memory or registers used for comparisons (Section 2.2). For instance, magic bytes found in headers are likely to take part in comparison instructions as-is or after some simple encoding transformation [5]. We apply DetectI2S to every operand in a comparison site instance and populate the $R$ data structure with newly discovered I2S facts.

MarkChecksums involves the detection of checksumming sequences. Similarly to RedQueen, we mark a comparison instruction as likely involved in a checksum if the following conditions hold: **(1)** one operand is I2S and its size is at least 2 bytes; **(2)** the other operand is not I2S and GetDeps revealed dependencies on some input bytes; and **(3)** $\bigwedge_b$(Deps(s, j, op$_1$)[b], Deps(s, j, op$_2$)[b]) = false, that is, the sets of their byte dependencies are disjoint.

The intuition is that code compares the I2S operand (which we speculate to be the expected value) against a value derived from input bytes, and those bytes do not affect the I2S operand or we would have a circular dependency. We choose a 2-byte minimum size to reduce false positives. Like prior works we patch candidate checksum tests to be always met, and defer to a later stage both the identification of false positives and the input repairing required to meet the condition(s) from the original unpatched program.

Finally, FuzzOperands replaces the deterministic mutations of AFL with surgical byte substitutions driven by data seen at comparison sites. For each input byte, we determine every CT entry (i.e., each operand of a comparison site instance) it influences. Then we replace the byte and, depending on the operand size, its surrounding ones using the value recorded for the other operand. The replacement can use such value as-is, rotate it for endianness, increment/decrement it by one, perform zero/sign extension, or apply ASCII encoding/decoding of digits. Each substitution yields an input added to the queue if its execution reveals a coverage improvement.

**Example.** When analyzing the function `parser`, Weizz is able to detect that the first operands of `CMP_A` and `CMP_B` and the second operand of `CMP_D` are I2S. On the other hand, the second operand of `CMP_C` and the first operand of `CMP_D` are not I2S, although they depend on some of the input bytes.

Weizz marks `CMP_D` as a comparison instance likely involved into a checksum since the three required conditions hold: the second operand is I2S and has size 2, the first operand is not I2S but depends on several input bytes, and the two operands show dependencies on disjoint sets of bytes.

FuzzOperands attempts surgical substitutions based on observed operand values: for instance, it can place as-is value `0xAAAA` recorded at the comparison site `CMP_A` in the bytes for the field `id` and trigger the enclosed `exit(1)` statement.

### 3.1.4 *Tag Placement.*

Weizz succinctly describes dependency information between input bytes and performed comparisons by annotating such bytes with tags essential for the subsequent structural information inference. For the $b$-th input byte $Tags[b]$ keeps:

- *id*: (the address of) the comparison instruction chosen as most significant among those $b$ influences;
- *ts*: the timestamp of the *id* instruction when first met;
- *parent*: the comparison instruction that led Weizz to the most recent tag assignment prior to $Tags[b]$;
- *dependsOn*: when $b$ contains a checksum value, the comparison instruction for the innermost nested checksum that verifies the integrity of $b$, if any;
- *flags*: stores the operand affected by $b$, if it is I2S, or if it is part of a checksum field;
- *numDeps*: the number of input bytes that the operand in *flags* depends on.

The tag assignment process relies on spatial and temporal locality in how comparison instructions get executed. Weizz tries to infer structural properties of the input based on the intuition that a program typically uses distinct instructions to process structurally distinct items. We can thus tag input bytes as related when they reach one same comparison directly or through derived data. When more candidates are found, we use temporal information to prioritize instructions with a lower timestamp, as input format validation normally takes place in parsing code from the early stages of the execution. As we explain next, we extend this scheme to account for checksums and to reassign tags when heuristics spot a likely better candidate than the current choice. Temporal information can serve also as proxy for hierarchical relationships with parent tags.

Algorithm PlaceTags iterates over comparison sites sorted by when first met in the execution, and attempts an inference for each input byte. We apply it to individual *op* instruction operands seen at a site $s$. If for the current byte $b$ we find no dependency among all recorded instances $Deps(s, j, op)$, the cycle advances to the next byte, otherwise we compute a *numDeps* candidate, i.e., the number $n$ of input bytes that affect the instruction, computed as $n \leftarrow \sum_k (1 \text{ if } \bigvee_j Deps(s, j, op)[k] \text{ else } 0)$ where $k$ indexes the input length. If the byte is untagged we tag it with the current instruction, otherwise we consider a reassignment. If the current tag $Tags[b]$ does not come from a checksum test and $n$ is smaller than $Tags[b].numDeps$, we reassign the tag as fewer dependencies suggest the instruction may be more representative for the byte.

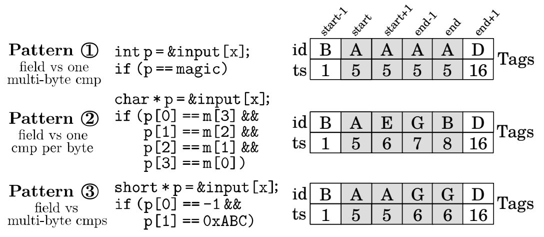
When we find a comparison treating the byte as from a checksum value, we always assign it as its tag. To populate the *dependsOn* field we use a topological sort of the dependencies *Deps* over each input byte, that is, we know when a byte part of a checksum value represents also a byte that some outer checksum(s) verify for integrity. We later repair inputs starting from the innermost checksum, with *dependsOn* pointing to the next comparison to process.

**Example.** Let us consider the seed of Figure 2c. The analysis of dependencies leads to the following tag assignments:

- bytes from field `id` affect sites `CMP_A` and `CMP_D`: `CMP_A` is chosen as tag as it is met first in the execution;
- bytes from `size` affect `CMP_B`, `CMP_C`, and `CMP_D`: `CMP_B` is chosen as tag as it temporally precedes the other sites;
- bytes from `data` and `checksum` affect only `CMP_D`, which becomes the *id* for their respective tags: however the tags will differ in the *flags* field as those for `checksum` bytes are marked as involved in a checksum field.

### 3.1.5 *Checksum Validation & Input Repair.*

The last surgical step involves checksum validation and input repair for checksums patched



```
int p = &input[x];
if (p == magic)
```

```
char * p = &input[x];
if (p[0] == m[3] &&
    p[1] == m[2] &&
    p[2] == m[1] &&
    p[3] == m[0])
```

```
short * p = &input[x];
if (p[0] == -1 &&
    p[1] == 0xABC)
```

**Figure 3: Field patterns identified by WEIZZ.**

in the program during previous iterations. As the technique is conceptually similar to the one of REDQUEEN, we discuss it briefly.

Algorithm FIXCHECKSUMS uses topologically sorted tags that were previously marked as involved in checksum fields. For each tag, it first extracts the checksum computed by the program (i.e., the input-derived operand value of the comparison) and where in the input the contents of the other operand (which is necessarily I2S, see Section 3.1.3) are stored, then it replaces such bytes with the expected value. It then disables the involved patch and runs the program on the repaired input: if it handled the checksum correctly, we will observe the same execution path as before, otherwise the checksum was a false positive and we have to bail.

At the end of the process WEIZZ re-applies patches that were not a false positive: this will benefit both the second stage and future surgical iterations over other inputs. It then also applies patches for checksums newly discovered by MARKCHECKSUMS in the current surgical stage, so when it will analyze again the input (or similar ones from FUZZOPERANDS) it will be able to overcome them.

## 3.2 Structure-Aware Stage

The second stage of WEIZZ generates new inputs via nondeterministic mutations that can build on tags assigned in the surgical stage. Like in the AFL realm, the number of derived inputs depends on an energy score that the power scheduler of AFL assigns to the original input [24]. Each input is the result of a variable number (1 to 256) of stacked mutations, with WEIZZ picking nondeterministically at each step a *havoc*, a *field*, or a *chunk* mutation scheme. The choice of keeping the havoc mutations of AFL is shared with previous chunk-oriented works [24], as combining them with structure-aware mutations improves the scalability of the approach.

Field mutations build on tags assigned in the surgical stage to selectively alter bytes that together likely represent a single piece of information. Chunk mutations target instead higher-level, larger structural transformations driven by tags.

As our field and chunk inference strategy is not sound, especially compared to manually written specifications, we give WEIZZ leeway in the identification process. WEIZZ never reconstructs the input structure in full, but only identifies individual fields or chunks in a nondeterministic manner when performing a mutation. The downside of this choice is that WEIZZ may repeat work or make conflicting choices among mutations, as it does not keep track of past choices. On the bright side, WEIZZ does not commit to possibly erroneous choices when applying one of its heuristics: this limits their impact to the subsequent mutations, and grants more leeway to the overall fuzzing process in exploring alternate scenarios.

```
function FIELDMUTATION(Tags, I):
1  b ← pick u.a.r. from {0 ... len(I)}
2  for i ∈ {b ... len(I)-1} do
3      if Tags[i].id==0 then continue
4      start ← GOLEFTWHILESAMETAG(Tags, i)
5      with probab. $Pr_{I2S}$ or if !I2S(Tags[start]) then
6          end ← FINDFIELDEND(Tags, start, 0)
7          I ← MUTATE(I, start, end); break
8  return I
```

**Algorithm 2:** Field identification and mutation.

We will describe next how we use input tags to identify fields and chunks and the mutations we apply. At the end of the process, we execute the program over the input generated from the stacked mutations, looking for crashes and coverage improvements. We promptly repair inputs that cause a new crash, while the others undergo repair when they eventually enter the surgical stage.

### 3.2.1 Fields.

Field identification is a heuristic process designed around prominent patterns that WEIZZ should identify. The first one is straightforward and sees a program checking a field using a single comparison instruction. We believe it to be the most common in real-world software. The second one instead sees a program comparing every byte in a field using a different instruction, as in the following fragment from the lodepng library:

```
unsigned char lodepng_chunk_type_equals(const unsigned char*
      chunk, const char* type) {
   if (strlen(type) != 4) return 0;
   return (chunk[4]==type[0] && chunk[5]==type[1] && chunk[6]==
        type[2] && chunk[7]==type[3]); }
```

which checks each byte in the input string chunk using a different comparison statement. Such code for instance often appears in program to account for differences in endianness. The two patterns may be combined to form a third one, as in the bottom part of Figure 3, that we consider in our technique as well.

Let us present how WEIZZ captures the patterns instantiated in Figure 3. For the first pattern, since a single instruction checks all the bytes in the field, we expect to see the corresponding input bytes marked with the same tag. For the second pattern, we expect instead to have consecutive bytes marked with different tags but consecutive associated timestamps, as no other comparison instruction intervenes. In the figure we can see a field made of bytes with tag ids {A, E, G, B} having respective timestamps {5, 6, 7, 8}. The third pattern implies having (two or more) subsequences made internally of the same tag, with the tag changing across them but with a timestamp difference of one unit only.

Procedure FIELDMUTATION (Algorithm 2) looks nondeterministically for a field by choosing a random position $b$ in the input. If there is no tag for the current byte, the cursor advances until it reaches a tagged byte (line 3). On the contrary, if the byte is tagged WEIZZ checks whether it is the first byte in the candidate field or there are any preceding bytes with the same tag, retracting to the leftmost in the latter case (line 4). With the start of the field now found, WEIZZ evaluates whether to mutate it: if the initial byte is I2S, the mutation happens only with a probability as such a byte could represent a magic number. Altering a magic number would lead to program paths handling invalid inputs, which in general are less appealing for a fuzzer. The extent of the mutation is decided by the helper procedure FINDFIELDEND, which looks for sequences of

```
struct { // lines
 int type; // 1-3
 int x, y; // 2-3
 int cksm; // 4-5
}
```

(a)

```
struct {
 int type; // 1-3
 int cksm; // 4-5
 int x, y; // 8-9
}
```

(b)

```
struct {
 int type; // 1-3
 int x, y; // 2-3
 int cksm; // 4-5
 char data[64]; // 7-9
}
```

(c)

(d)

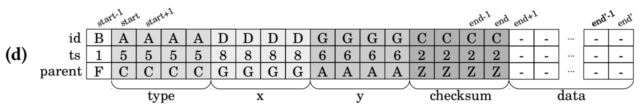


**Figure 4: Examples of chunks that Weizz can look for.**

```
function FindChunkEnd(Tags, k):
1  end ← GoRightWhileSameTag(Tags, k)
2  while Tags[end+1].ts >= Tags[k].ts do
3      end ← FindChunkEnd(Tags, end+1)
4  while Tags[end+1].id == Tags[k].parent do
5      end ← end +1
6  with probability $Pr_{extend}$ do
7      while Tags[end+1].id == 0 do end ← end +1
8      while Tags[end+1].ts >= Tags[k].ts do
9          end ← FindChunkEnd(Tags, end+1)
10 return end
```

**Algorithm 3:** Boundary identification for chunks.

tagged bytes that meet one of the three patterns discussed above. FieldMutation then alters the field by choosing one among the twelve AFL length-preserving havoc transformations over its bytes.

### 3.2.2 *Chunks.*

Weizz heuristically locates plausible boundaries for chunks in the input sequence, then it applies higher-order mutations to candidate chunks. As running example we discuss variants of a structure representative of many chunks found in binary formats.

The first variant (Figure 4a) has four fields: a `type` assigned with some well-known constant, two data fields `x` and `y`, and a `cksm` value for integrity checking. Let us consider a program that first computes the checksum of the structure and compares it against the namesake field, then it verifies the `type` and `x` fields in this order, executes possibly unrelated comparisons (i.e., they do not alter tags for the structure bytes), and later on makes a comparison depending on `y`. For the sake of simplicity we assume that field processing matches the first pattern seen in the previous section (i.e., one comparison per field). The output of PlaceTags will be coherent with the graphical representation of Figure 4d.

We now describe how our technique is able to identify the boundaries of this chunk using tags and their timestamps. Similarly as with fields, we pick a random position *b* within the input and analyze the preceding bytes, retracting the cursor as long as the tag stays the same. For any *b* initially falling within the 4 bytes of `type`, Weizz finds the same *start* index for the chunk. To find the end position it resorts to FindChunkEnd (Algorithm 3).

The procedure initially recognizes the 4 bytes of `type` (line 1), then recursively looks for adjacent fields when their timestamps are higher than the one for the current field (lines 2-3). This recursive step reveals fields `x` and `y`. The `cksm` field has a lower timestamp value (as the program checked it before analyzing `type`), but lines 4-5 can include it in the chunk as they inspect parent data. The parent is the comparison from the tag assignment prior to the current one, and the first activation of FindChunkEnd will see that the tag of the first byte of `cksm` matches the parent for the tag for `type`.

To explain lines 6-9 in Algorithm 3, let us consider variants of the structure that exercise them. In Figure 4b `cksm` comes before `type`, and in this case the algorithm would skip over lines 2-3 without incrementing *end*, thus missing `x` and `y`. Lines 8-9 can add to the chunk bytes from adjacent fields as long as they have increasing timestamps with respect to the one from the tag for the *k*-th byte (the first byte in `type` in this case). In Figure 4c we added an array `data` of 64 bytes to the structure: it may happen that Weizz leaves binary blobs untagged if the program does not make comparisons over them or some derived values. Line 7 can add such sequences to a chunk, extending *end* to the new *end′* value depicted in Figure 4d.

With FindChunkEnd we propose an inference scheme inspired by the layout of popular formats. The first field in a chunk is typically also the one that the program analyzes first, and we leverage this fact to speculate where a chunk may start. If the program verifies a checksum before accessing even that field, we link it to the chunk using parent information, otherwise "later" checksums represent a normal data field. Lines 7-9 enlarge chunks to account for different layouts and tag orderings, but only with a probability to avoid excessive extension. The algorithm can also capture partially tagged blobs through the recursive steps it can take.

Observe however that in some formats there might be dominant chunk types, and choosing the initial position randomly would reveal a chunk of non-dominant type with a smaller probability. We devise an alternate heuristic that handles this scenario better: it randomly picks one of the comparison instructions appearing in tags, assembles non-overlapping chunks with FindChunkEnd starting at a byte tagged by such an instruction, and picks one among the built chunks. As Weizz is unaware of the format characteristics, we choose between the two heuristics with a probability.

For the current chunk selection, Weizz attempts one of the following higher-order mutations (Section 2.3):

- **Addition.** It adds a chunk from another input to the chunk that encloses the current one. Weizz picks a tagged input I′ from the queue and looks for chunks in it starting with bytes tagged with the same parent field of the leading bytes in the current chunk. It picks one randomly and extends the current input by adding its associated bytes before or after the current chunk. The parent tag acts as proxy for the nesting information available instead to AFLSmart.
- **Deletion**. It removes the input bytes for the chunk.
- **Splicing.** It picks a similar chunk from another input to replace the current one. It scans that input looking for chunks starting with the same tag (AFLSmart uses type information) of the current, randomly picks one, and substitutes the bytes.

### 3.2.3 *Discussion.*

We can now elaborate on why, in order to back the field and chunk mutations just described, we cannot rely on the dependency identification techniques of RedQueen or SLF.

Let us assume RedQueen colors an input with a number of `A` bytes and initially logs a `cmp A, B` operation. RedQueen would attempt to replace each occurrence of `A` with `B` and validate it by looking for coverage improvements when running the program over the new input, this time with logging disabled. This strategy works well if the goal are only I2S replacements, but for field identification there are two problems: (i) with multiple `B` bytes in the input, we cannot determine which one makes a field for the comparison, and (ii) if B is already in another input in the queue, no coverage improvement comes from the replacement, and RedQueen misses a direct dependency for the current input. Our structural mutations require tracking comparisons all along for I2S and non-I2S facts.

SLF can identify dependencies similarly to our GetDeps, but recognizes only specialized input portions—that we can consider fields—based on how the program treats them. SLF supports three categories of program checks and can mutate portions e.g. by replicating those involved in a count check (§2.2). While Weizz may do that through chunk addition, it also mutates fields—and whole chunks—that are not treated by SLF.

## 4 IMPLEMENTATION

We implemented our ideas on top of AFL 2.52b and QEMU 3.1.0 for x86-64 Linux targets. For branch coverage and comparison tables we use a shadow call stack to compute context-sensitive information [14], which may let a fuzzer explore programs more pervasively [10]. We index the coverage map using the source and destination basic block addresses and a hash of the call stack.

Natural candidates for populating comparison tables are `cmp` and `sub` instructions. We store up to $|J|$=256 entries per site. Like RedQueen we also record `call` instructions that may involve comparator functions: we check whether the locations for the first two arguments contain valid pointers, and dump 32 bytes for each operand. Treating such calls as operands (think of functions that behave like `memcmp`) may improve the coverage, especially when the fuzzer is configured not to instrument external libraries.

An important optimization involves *deferring* surgical fuzzing with a timeout-based mechanism, letting inputs jump to the second stage with a decreasing probability if an interesting input was discovered in the current window (sized as 50 seconds).

Note that untagged inputs can only undergo havoc mutations in the second stage: we thus introduce *derived* tags, which are an educated guess on the actual tag based on tags seen in a similar input. Derived tags speed up our fuzzer, and actual tags replace them when the input eventually enters the surgical stage.

Two scenarios can give rise to derived tags. One case happens when starting from an input I the surgical mutations of FuzzOperands(I) produce one or more inputs I' that improve coverage and are added to the queue. Once PlaceTags has tagged I, we copy such tags to I' (structurally analogous to I, as FuzzOperands operates with "local" mutations) and mark them as derived.

Similarly, a tagged input I1 can undergo high-order mutations that borrow bytes from a tagged input I2, namely addition or splicing. In this case the added/spliced bytes of the mutated I1' coming from I2 get the same tags seen in I2, while the remaining bytes keep the tags seen for them in I1.

## 5 EVALUATION

In our experiments we tackle the following research questions:

- **RQ 1.** How does Weizz compare to state-of-the-art fuzzers on targets that process chunk-based formats?
- **RQ 2.** Can Weizz identify new bugs?
- **RQ 3.** How do tags relate to the actual input formats? And what is the role of structural mutations and roadblock bypassing in the observed improvements?

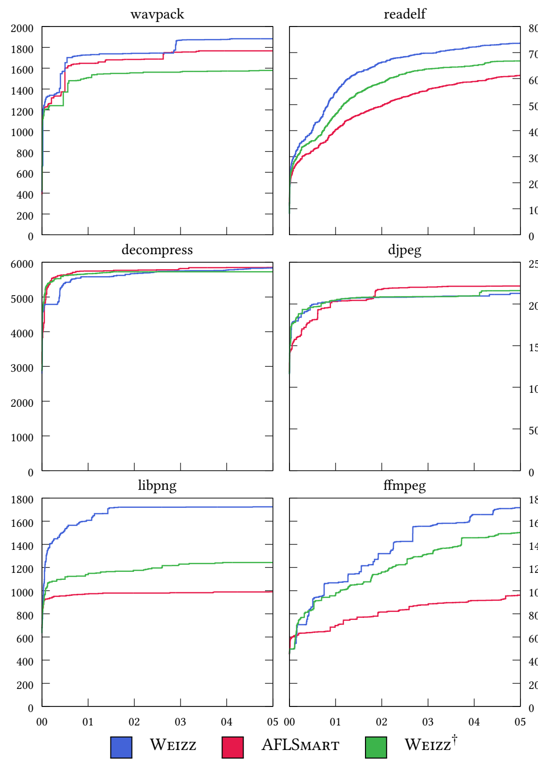


**Figure 5: Basic block coverage over time (5 hours).**

**Benchmarks.** We consider the following programs (version, input format): wavpack (5.1.0, WAV), decompress (2.3.1, JP2), ffmpeg (4.2.1, AVI), libpng (1.6.37, PNG), readelf (2.3.0, ELF), djpeg (5db6a68, JPEG), objdump (2.31.51, ELF), mpg321 (MP3, 0.3.2), oggdec (1.4.0, OGG), tcpdump (4.9.2, PCAP), and gif2rgb (5.2.1, GIF). The first 6 programs were considered in past evaluation of AFLSmart, with a format specification available for it. The last 8 programs are commonly used in evaluations of general-purpose fuzzers.

**Experimental setup.** We ran our tests on a server with two Intel Xeon E5-4610v2@2.30GHz CPUs and 256 GB of RAM, running Debian 9.2. We measured the cumulative basic block coverage of different fuzzers from running an application over the entire set of inputs generated by each fuzzer. We repeated each experiment 5 times, plotting the median value in the charts. For the second stage of Weizz we set $\Pr_{field} = \Pr_{chunk} = 1/15$ (Figure 1), similarly to the probability of applying smart mutations in AFLSmart in [24].

### 5.1 RQ1: Chunk-Based Formats

We compare Weizz against the state-of-the-art solution for chunk-based formats AFLSmart, which applies higher-order mutations over a virtual input structure (Section 2.3). We then take into account general-purpose fuzzers that previous studies [11, 24] suggest as being still quite effective in practice on this type of programs.

#### 5.1.1 *AFLSmart.*

For AFLSmart we used its release `604c40c` and the peach pits written by its authors for the virtual input structures involved. We measured the code coverage achieved by: (a) AFLSmart with stacked mutations but without deterministic stages as suggested in its documentation, (b) Weizz in its standard config-

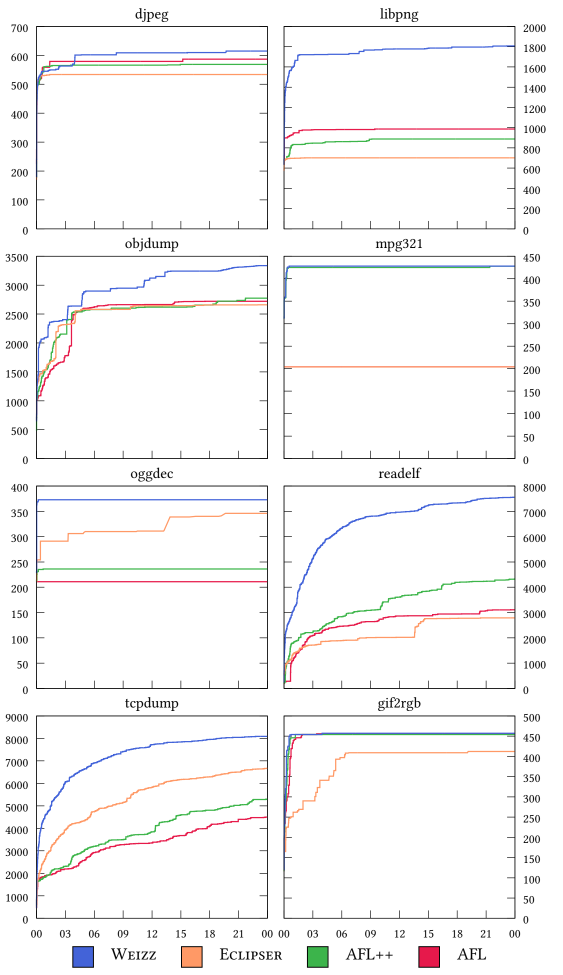


**Figure 6: Basic block coverage over time (24 hours).**

uration, and (c) a variant Weizz† with FuzzOperands, checksum patching, and input repairing disabled. Weizz† lets us focus on the effects of tag-based mutations, giving up on roadblock bypassing capabilities missing in AFLSmart. We provide (a) and (c) with a dictionary of tokens for format under analysis like in past evaluations of AFLSmart. We use AFL test cases as seeds, except for wavpack and ffmpeg where we use minimal syntactically valid files [9].

Figure 5 plots the median basic block coverage after 5 hours. Compared to AFLSmart, Weizz brings appreciably higher coverage on 3 out of 6 subjects (readelf, libpng, ffmpeg), slightly higher on wavpack, comparable on decompress, and slightly worse on djpeg. To understand these results, we first consider where Weizz† lies, and then discuss the benefits from the additional features of Weizz.

Higher coverage in Weizz† comes from the different approach to structural mutations. AFLSmart follows a format specification, while we rely on how a program handles the input bytes: Weizz† can reveal behaviors that characterize the actual implementation and that may not be necessarily anticipated by the specification. The first implication is that Weizz† mutates only portions that the program has already processed in the execution, as tags derive from executed comparisons. The second is that imprecision in inferring structural facts may actually benefit Weizz†. The authors of AFLSmart acknowledge how relaxed specifications can expose imprecise implementations [24]: we will return to this in Section 6.

Weizz† is a better alternative than AFLSmart for readelf, libpng, and ffmpeg. When we consider the techniques disabled in Weizz†, Weizz brings higher coverage for two reasons. I2S facts help Weizz replace magic bytes only in the right spots compared to dictionaries, which can also be incomplete. This turns out important also in multi-format applications like ffmpeg. Then, checksum bypassing allows it to generate valid inputs for programs that check data integrity, such as libpng that computes CRC-32 values over data.

We also consider the collected crashes, found only for wavpack and ffmpeg. In the first case, the three fuzzers found the same bugs. For ffmpeg, Weizz found a bug from a division by zero in a code section handling multiple formats: we reported the bug and its developers promptly fixed it. The I2S-related features of Weizz were very effective for generating inputs that deviate significantly from the initial seed, e.g., mutating an AVI file into an MPEG-4 one. In Section 5.2 we back this claim with one case study.

### 5.1.2 *Grey-Box Fuzzers.*

We now compare Weizz against 8 popular applications handling chunk-based formats, heavily tested by the community [2], and used in past evaluations of state-of-the-art fuzzers [5, 21, 24, 25]. We tested the following fuzzers: (a) AFL 2.53b in QEMU mode, (b) AFL++ 2.54c in QEMU mode enabling the CompareCoverage feature, and (c) Eclipser with its latest release when we ran our tests (commit `8a00591`) and its default configuration. As Weizz targets binary programs, we rule out fuzzers like Angora that require source instrumentation. For sub-instruction profiling, since Steelix is not publicly available, we choose AFL++ instead.

While considering RedQueen could be tantalizing, its hardware-assisted instrumentation could make an unfair edge as here we compare coverage of QEMU-based proposals, as the different overheads may mask why a given methodology reaches some coverage. We attempt an experiment in Section 5.3 by configuring Weizz to resemble its features, and compare the two techniques in Section 6.

As the competitors we consider have no provisions for structural mutations, we opted for a larger budget of 24 hours in order to see whether they could recover in terms of coverage over time. Consistently with previous evaluations [11], we use as initial seed a string made of the ASCII character "0" repeated 72 times. However, for libpng and tcpdump we fall back to a valid initial input test (`not_kitty.png` from AFL for libpng and a PCAP of a few seconds for tcpdump) as no fuzzer, excluding Weizz on libpng, achieved significant coverage with artificial seeds. We also provide AFL with format-specific dictionaries to aid it with magic numbers.

Figure 6 plots the median basic block coverage over time. Weizz on 5 out 8 targets (libpng, oggdec, tcpdump, objdump and readelf) achieves significantly higher code coverage than other fuzzers. The first three process formats with checksum fields that only Weizz repairs, although Eclipser appears to catch up over time for oggdec. Structural mutation capabilities combined with this factor may explain the gap between Weizz and other fuzzers. For objdump and readelf, we may speculate I2S facts are boosting the work of Weizz by the way RedQueen outperforms other fuzzers over them in its evaluation [5] (in objdump logging function arguments was crucial [5]). On mpg321 and gif2rgb, AFL-based fuzzers perform very similarly, with a large margin over EcLIPSER, confirming that standard AFL mutations can be effective on some chunk-based formats. Finally, WEIZZ leads for djpeg but the other fuzzers are not too far from it. Overall, AFL is interestingly the best alternative to WEIZZ for djpeg, libpng, and gif2rgb. When taking into account crashes, WEIZZ and AFL++ generated the same crashing inputs for mpg321, while only WEIZZ revealed a crash for objdump.

## 5.2 RQ2: New Bugs

To explore its effectiveness, we ran WEIZZ for 36 hours on several real-world targets, including the processing of inputs not strictly adhering to the chunk-based paradigm. WEIZZ revealed 16 bugs in 9 popular, well-tested applications: objdump, CUPS (2 bugs), libmirage (2), dmg2img (3), jbig2enc, mpg321, ffmpeg (3 in libavformat, 1 in libavcodec), sleuthkit, and libvmdk. Overall 6 bugs are NULL pointer dereferences (CWE-476), 1 involves an unaligned realloc (CWE-761), 2 can lead to buffer overflows (CWE-122), 2 cause an out-of-bounds read (CWE-125), 2 a division by zero (CWE-369), and 3 an integer overflow (CWE-190). We detail two interesting ones.

**CUPS.** While the HTML interface of the Common UNIX Printing System is not chunk-oriented, we explored if WEIZZ could mutate the HTTP requests enclosing it. WEIZZ crafted a request that led CUPS to reallocate a user-controlled buffer, paving the road to a *House of Spirit* attack [7]. The key to finding the bug was to have FUZZOPERANDS replace some current input bytes with I2S facts (`'Accept-Language'` logged as operand in a call to `_cups_strcasecmp`), thus materializing a valid request field that chunk mutations later duplicated. Apple confirmed and fixed the bug in CUPS v2.3b8.

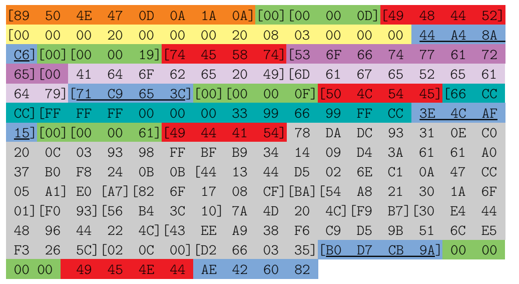
**libMirage.** We found a critical bug in a library providing uniform access to CD-ROM image formats. An attacker can generate a heap-based buffer overflow that in turn may corrupt allocator metadata, and even grant root access as the CDEmu daemon using it runs with root privileges on most Linux systems. We used an ISO image as initial seed: WEIZZ exposed the bug in the NRG image parser, demonstrating how it can deviate even considerably among input formats based exclusively on how the code handles input bytes.

## 5.3 RQ3: Understanding the Impact of Tags

Programs can differ significantly in how they process input bytes of different formats, yet we find it interesting to explore *why* WEIZZ can be effective on a given target. We discuss two case studies where we seek how tags can assist field and chunk identification in libpng, and we see how smart mutations and roadblocks bypassing are essential for ffmpeg, but either alone is not enough for efficacy.

**Tag Accuracy.** Figure 7 shows the raw bytes of the seed `not_kitty.png` for libpng. It starts with a 8-byte magic number (in orange) followed by 5 chunks, each characterized by four fields: length (4 bytes, in green), type (4 bytes, red), data (heterogeneous, as with `PNG_CHUNK_IHDR` data in yellow), and a CRC-32 checksum (4 bytes, in blue). The last chunk has no data as it marks the file end.

To understand the field identification process, we analyzed the tags from the surgical stage for the test case. Starting from the first byte, we apply FINDFIELDEND repeatedly at every position after the end of the last found field. In Figure 7 we delimit each found field in square brackets, and underline bytes that WEIZZ deems from



```
[89 50 4E 47 0D 0A 1A 0A][00][00 00 0D][49 48 44 52]
[00 00 00 20 00 00 00 20 08 03 00 00 00 44 A4 8A
C6][00][00 00 19][74 45 58 74][53 6F 66 74 77 61 72
65][00 41 64 6F 62 65 20 49][6D 61 67 65 52 65 61
64 79][71 C9 65 3C][00][00 00 0F][50 4C 54 45][66 CC
CC][FF FF FF 00 00 00 33 99 66 99 FF CC 3E 4C AF
15][00][00 00 61][49 44 41 54] 78 DA DC 93 31 0E C0
20 0C 03 93 98 FF BF B9 34 14 09 D4 3A 61 61 A0
37 B0 F8 24 0B 0B [44 13 44 D5 02 6E C1 0A 47 CC
05 A1] E0 [A7][82 6F 17 08 CF][BA][54 A8 21 30 1A 6F
01][F0 93][56 B4 3C 10] 7A 4D 20 4C][F9 B7][30 E4 44
48 96 44 22 4C][43 EE A9 38 F6 C9 D5 9B 51 6C E5
F3 26 5C][02 0C 00][D2 66 03 35][B0 D7 CB 9A] 00 00
00 00 49 45 4E 44 AE 42 60 82
```

`[]`: field delimiters from WEIZZ <u>HEX</u>: checksum field from WEIZZ

PNG fields: 🟧 PNG_SIGNATURE 🟩 LENGTH 🟥 TYPE 🟨 PNG_CHUNK_IHDR 🟦 CRC-32 ⬜ IDAT DATA 🟪 PNG_CHUNK_TEXT LABEL ▨ PNG_CHUNK_TEXT DATA 🟦 PNG_CHUNK_PLTE

**Figure 7: Fields identified in the `not_kitty.png` test case.**

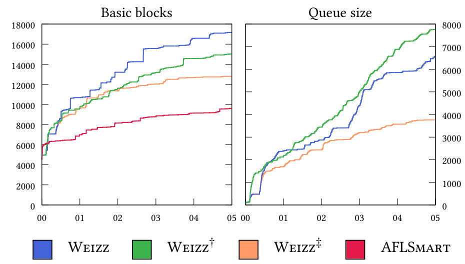


**Figure 8: Analysis of ffmpeg with three variants of WEIZZ.**

checksum fields. WEIZZ identifies correct boundaries with a few exceptions. The last three fields in the file are not revealed, as libpng never accesses that chunk. In some cases a data field is merged with the adjacent checksum value. Initially libpng does not make comparisons over the data bytes, but first computes their checksum and verifies it against the expected value with a comparison that will characterize also the data bytes (for the data-flow dependency). Dependency analysis on the operands (Section 3.1.3) however identifies the checksum correctly for FIXCHECKSUMS, and once the input gets repaired and enters the surgical stage again, libpng will execute *new* comparisons over the data bytes, and WEIZZ can identify the data field correctly tagging them with such instructions. Finally, the grey parts (`IDAT DATA`) represent a binary blob fragmented into distinct fields by WEIZZ with gaps from untagged bytes, as not all the routines that process them make comparisons over the blob.

Analyzing chunk boundaries is more challenging, as FINDFIELDEND is sensitive to the initial position. For instance, when starting the analysis at the 8-byte magic number it correctly identifies the entire test case as one chunk, or by starting at a length field it may capture all the other fields for that chunk[3]. However, when starting at a type field it may build an incomplete chunk. Nonetheless, even under imprecise boundary detection WEIZZ can still make valuable mutations for two reasons. First, mutations that borrow bytes from other test cases check the same leading tag, and this yields for instance splicing of likely "compatible" incomplete chunks. Second, this may be beneficial to exercise not so well-tested corner cases or shared parsing code: we will return to it in Section 6. For libpng we achieved better coverage than other fuzzers, including AFLSMART.

[3]Depending on the starting byte, it could miss some of the first bytes.

**Structural Mutations vs. Roadblocks.** To dig deeper in the experiments of Section 5.1, one question could be: *how much coverage improvement is due to structural mutations or roadblock bypassing?* We consider ffmpeg as case study: in addition to Weizz and Weizz$^\dagger$ that lacks roadblock bypassing techniques but leverages structural mutations, we introduce a variant Weizz$^\ddagger$ that can use I2S facts to bypass roadblocks like RedQueen but lacks tag-based mutations. In Figure 8 we report code coverage and input queue size after a run of 5 hours[4]. When taken individually, tag-based mutations seem to have an edge over roadblock bypass techniques (+17% coverage). This may seem surprising as ffmpeg supports multiple formats and I2S facts can speed up the identification of their magic numbers, unlocking several program paths. Structural mutations however affect code coverage more appreciably on ffmpeg. When combining both features in Weizz, we get the best result with 34% more coverage than in Weizz$^\ddagger$. Interestingly, Weizz$^\dagger$ shows a larger queue size than Weizz: this means that although Weizz explores more code portions, Weizz$^\dagger$ covers some of the branches more exhaustively, i.e., it is able to cover more hit count buckets for some branches.

## 6 CONCLUDING REMARKS

Weizz introduces novel ideas for computing byte dependency information to simultaneously overcome roadblocks and back fully automatic fuzzing of chunk-based binary formats. The experimental results seem promising: we are competitive with human-assisted proposals, and we found new bugs in well-tested software.

Our approach has two practical advantages: fuzzers already attempt bitflips in deterministic stages, and instrumenting comparisons is becoming a common practice for roadblocks. We empower such analyses to better characterize the program behavior while fuzzing, enabling the tag assignment mechanism. Prior proposals do not offer sufficient information to this end: even for RedQueen, its colorization [5] identifies I2S portions of an input (crucial for roadblocks) but cannot reveal dependencies for non-I2S bytes.

A downside is that bit flipping can get costly over large inputs. However, equally important is the time the program takes to execute one test case. In our experiments Weizz applied the surgical stage to inputs up to 3K bytes, with comparable or better coverage than the other fuzzers we considered. We leave to future work using forms of bit-level DTA [32, 33] as an alternative for "costly" inputs.

Weizz may miss dependencies for comparisons made with uninstrumented instructions. This can happen in optimized code that uses arithmetic and logical operations to set CPU flags for a branch decision. We may resort to intra-procedural static analysis to spot them [25] (as logging all of them blindly can be expensive) but currently opted for tolerating some inconsistencies in the heuristics we use, for instance skipping over one byte in FindChunkEnd when the remaining bytes would match the expected patterns.

Our structure-aware stage, in addition to not requiring a specification, is different than AFLSmart also in where we apply high-order operators. AFLSmart mutates chunks in a black-box fashion, that is, it has no evidence whether the program manipulated the involved input portion during the execution. Weizz chooses among chunks that the executed comparison instructions indirectly reveal. We find hard to argue which strategy is superior in the general case. Another important difference is that, as our inference schemes are not sound, we may mutate inputs in odd ways, for instance replacing only portions of a chunk with a comparable counterpart from another input. In the AFLSmart paper the authors explain that they could find some bugs only thanks to a relaxed specification [24]. We find this consistent with the Grimoire experience with grammars, where paths outside the specification revealed coding mistakes [8].

As future work we plan to consider a larger pool of subjects, and shed more light on the impact of structure-aware techniques: how they impact coverage, which are the most effective for a program, and how often the mutated inputs do not meet the specification. Answering these questions seems far from trivial due to the high throughput and entropy of fuzzing. There is also room to extend chunk inference with new heuristics or make the current ones more efficient. For instance, we are exploring with promising results a variant where we locate the beginning of a chunk at a field made of I2S bytes, possibly indicative of a magic value for its type.

[4]AFLSmart shown as reference—queues are uncomparable (no context sensitivity).

## REFERENCES

[1] 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. [Online; accessed 10-Sep-2019].

[2] 2019. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed 10-Sep-2019].

[3] Dave Aitel. 2002. The Advantages of Block-Based Protocol Analysis for Security Testing. https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html. [Online; accessed 10-Sep-2019].

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018), 39 pages. https://doi.org/10.1145/3182657

[7] Blackngel. 2019. MALLOC DES-MALEFICARUM. http://phrack.org/issues/66/10.html. [Online; accessed 10-Sep-2019].

[8] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002. https://www.usenix.org/system/files/sec19-blazytko.pdf

[9] Mathias Bynens. 2019. Smallest possible syntactically valid files of different types. https://github.com/mathiasbynens/small. [Online; accessed 10-Sep-2019].

[10] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[11] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 736–747. https://doi.org/10.1109/ICSE.2019.00082

[12] 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 (CCS '08)*. 391–402. https://doi.org/10.1145/1455770.1455820

[13] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. 15–27. https://doi.org/10.1145/3321705.3329819

[14] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. Mining Hot Calling Contexts in Small Space. *Software: Practice and Experience* 46 (2016), 1131–1152. https://doi.org/10.1002/spe.2348

[15] M. Eddington. [n.d.]. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 10-Sep-2019].

[16] Marc Heuse, Heiko Eißfeldt, and Andrea Fioraldi. 2019. AFL++. https://github.com/vanhauser-thc/AFLplusplus. [Online; accessed 10-Sep-2019].
[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (SEC'12)*. 38–38. http://dl.acm.org/citation.cfm?id=2362793.2362831
[18] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 720–725. https://doi.org/10.1145/2970276.2970321
[19] Matthias Höschele and Andreas Zeller. 2017. Mining Input Grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. 31–34. https://doi.org/10.1109/ICSE-C.2017.14
[20] Mateusz Jurczyk. 2019. CompareCoverage. https://github.com/googleprojectzero/CompareCoverage/. [Online; accessed 10-Sep-2019].
[21] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 627–637. https://doi.org/10.1145/3106237.3106295
[22] Tavis Ormandy. 2009. Making Software Dumberer. http://taviso.decsystem.org/making_software_dumber.pdf. [Online; accessed 10-Sep-2019].
[23] H. Peng, Y. Shoshitaishvili, and M. Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056
[24] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681
[25] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/
[26] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*. 317–331. https://doi.org/10.1109/SP.2010.26
[27] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *23th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
[28] Robert Swiecki. 2017. honggfuzz. https://github.com/google/honggfuzz. [Online; accessed 10-Sep-2019].
[29] Ari Takanen, Jared D. Demott, and Charles Miller. 2018. *Fuzzing for Software Security Testing and Quality Assurance* (2nd ed.). Artech House, Inc., Norwood, MA, USA.
[30] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081
[31] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy (SP)*. 497–512. https://doi.org/10.1109/SP.2010.37
[32] B. Yadegari and S. Debray. 2014. Bit-Level Taint Analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 255–264. https://doi.org/10.1109/SCAM.2014.43
[33] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy (SP)*. 674–691. https://doi.org/10.1109/SP.2015.47
[34] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080
[35] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. 745–761. http://dl.acm.org/citation.cfm?id=3277203.3277260
[36] Michał Zalewski. 2019. American Fuzzy Lop. https://github.com/Google/AFL. [Online; accessed 10-Sep-2019].
[37] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. 2019. The Fuzzing Book. https://www.fuzzingbook.org/. [Online; accessed 10-Sep-2019].