WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats

Andrea Fioraldi  
Sapienza University of Rome  
andrea.fioraldi@gmail.com

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

Emilio Coppa  
Sapienza University of Rome  
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, in this paper we present a technique to automatically generate and mutate inputs for unknown chunk-based binary formats. We propose a technique to identify dependencies between input bytes and comparison instructions, and later 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.

We show that our techniques performs comparably to structure-aware fuzzing proposals that require human assistance. Our prototype implementation WEIZZ revealed 11 unknown bugs in widely used programs.

1. Introduction

Recent years have witnessed a spike of activity in the development of efficient techniques for fuzz testing, also known as fuzzing, from both academic and industrial actors. 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 [1] 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 [2], it can lose efficacy when dealing with 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 such a 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 [2], [3] and chunk-based [4] formats.

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 [6].

The second limitation is that testing only inputs that perfectly adhere to the specification would miss imprecisions in the implementation, while inputs that are to some degree outside it may instead exercise them [6]. In our experience such bugs are present also in applications that operate on multiple input formats and share code for their processing.

Recently Blazytko et al. [6] have shown in the context of grammar-based formats how to extend the grey-box fuzzing process to resemble the workings of a grammar-based solution without requiring the user to supply one. In a similar vein in our work we explore automatic fuzzing of chunk-based structures found in many binary formats [4], [7].

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

Nuances of this idea are present in [6] where code coverage guides grammar structure inference, and to some extent also in a few general-purpose fuzzing additions. For instance, taint tracking can reveal which input bytes take part in comparisons against magic sequences [8], [9]. Similarly, input-to-state correspondence [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 operations for their processing, and devise techniques to infer plausible boundaries for chunks and fields. Thanks to this automatic pipeline we can apply structure-aware mutations for chunk-based grey-box fuzzing [4] 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: we leverage the fact that a program typically uses distinct instructions in the form of comparisons to parse distinct items; to break ties we prioritize older instructions since 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 scenarios for reconstructing specifications of input formats [7], [10], [11], experimental results suggest that, like observed for grammar-based formats [6], this 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, but also devise optimized variants of one functionality, for instance specialized on 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.

The contributions of this work can be summarized as:
- a dependency identification technique embedded in the deterministic mutation stage of grey-box fuzzing;
- a tag assignment mechanism that uses dependencies to overcome roadblocks and to back a structure inference scheme for chunk-based formats;
- a prototype implementation of the approach on top of QEMU and AFL that we call WEIZZ;
- an experimental investigation where WEIZZ beats or matches a chunk-based CGF proposal that accesses a specification, and outperforms general-purpose fuzzers over the applications we considered;
- a discussion of some real-world bugs we discovered.

To facilitate analyses and extensions of our approach, we will make WEIZZ available as open source.

2. State of the Art

The last years have seen a large body of fuzzing-related works [12]. 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 [13]. 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 [4], yet it may struggle in the presence of roadblocks (like checksum and magic numbers) [5], or when mutating structured grammar-based [6] or chunk-based [4] input formats. In the following 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) [1] 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 $2^{16}$ entries and associates a counter to the hash of the branch, built using its source and destination basic block addresses. AFL considers 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 to power-of-two buckets before lookup.

Mutations. In the deterministic stage AFL sequentially scans the input, applying a set of mutations for each position. These include 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 the next transformation.

In the havoc stage 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. AFL attempts alternate mutation sequences on the currently selected input: a power scheduler chooses the number of inputs to generate at this stage based on the program running time and the size of the generated coverage trace over the single input.

2.2. Roadblocks

In the fuzzing jargon 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. In the Appendix we report examples for both from the chunk parsing code of a PNG processing software (Section A.1).

Format-specific dictionaries may help with magic numbers, yet the fuzzer has to figure out where to place such values. Researchers thus came up with a few approaches to handle magic numbers and checksums in grey-box fuzzers.

Sub-instruction Profiling. While there is no obvious way to figure out how a large amount of logic can be encoded in a single comparison, one could break multi-byte comparisons into single-byte [14] (or even single-bit [15]) checks to better track progress when attempting to match constant values. LAF-INTEL [14] and COMPARECOVERAGE [16] are compiler extensions to produce binaries for
such a fuzzing. HONGGFUZZ [17] implements this technique for fuzzing programs when the source is available, while AFL++ [18] can automatically transform binaries during fuzzing. STEELIX [19] resorts to static analysis to filter out likely uninteresting comparisons from profiling. Overall, sub-instruction profiling is valuable to overcome tests against magic numbers, but ineffective with checksums.

**Taint Analysis.** Dynamic taint analysis (DTA) [20] can track when and which input parts affect program instructions. VUZZER [8] uses DTA on binary programs to bypass branches that check magic numbers, 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 [9] brings two technical improvements: it can identify magic numbers that are not contiguous in the input thanks to 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.** While effective for magic bytes, DTA techniques can at best identify checksum tests, but not provide sufficient information to address them. For this reason several proposals (e.g., [21], [22], [23], [24]) use symbolic execution [25] to identify and try to solve complex input constraints involved in generic roadblocks. TAINTSCOPE [23] proposes a methodology to identify possible checksums with DTA, patch them away for the sake of fuzzing, and later try to repair inputs with symbolic execution. The technique requires the user to provide specific seeds and is subject to false positives [21], [24].

Yet input constraints may be just too hard or long to solve, leading to timeouts without any actual progress. TFUZZ [24] divides checks in two categories: non-critical, when they involve magic numbers, and critical, when they describe checksums, suggesting symbolic execution to deal with the first kind, and manual analysis for the second.

DRILLER [21] follows a different design using symbolic execution as an alternate strategy to explore inputs, switching to it when fuzzing exhausts a time budget obtaining no coverage improvement. QSYM [22] adopts a similar scheme, choosing concolic execution implemented via dynamic binary instrumentation [26] 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 without incurring their overheads. REDQUEEN [5] builds on the observation that often input bytes flow directly, or after simple encodings (e.g. swaps for endianness), into instruction operands. The authors show how to use this input-to-state correspondence instead of DTA and symbolic execution to deal with magic bytes, multi-byte compares, and checksums. REDQUEEN in particular approximates DTA by changing input bytes with random values (colorization) to see which comparison operands change, suggesting a dependency. When both operands of a comparison change, but only for one there is an input-to-state 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.

In the context of concolic fuzzers, ECLIPSER [27] relaxes the path constraints over each input byte. It runs a program multiple times 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 [25].

### 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. Bringing format awareness into play can boost CGF in this setting. In the literature we can distinguish techniques targeting either grammar-based formats, where inputs comply to a language grammar, or chunk-based ones, where inputs follow a tree organization with data chunks resembling a C structure to form individual nodes. The latter category seems prevalent in real-world software [4].

**Grammar-based Fuzzing.** LANGFUZZ [28] is a black-box fuzzer that using a Javascript grammar specification generates valid inputs for a language interpreter by combining sample code fragments (taken, e.g., from bug reports) and test cases. NAUTILUS [3] and SUPERION [2] are recent grey-box fuzzer proposals that can effectively test language interpreters without requiring a large corpus of valid inputs or fragments, but only an ANTLR grammar file.

GRIMOIRE [6] 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.** In the context of black-box techniques, SPIKE [29] lets users describe the network protocol in use for an application to improve the chance of finding bugs. PEACH [30] 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 blackbox fuzzers smart [4]. One could argue that a smart grey-box variant would outperform them, as due to the lack of feedback (as in explored code) they do not keep mutating interesting inputs.

AFLSMART [4] 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
The nondeterministic working of AFL with structure-aware techniques. WEIZZ builds on tags to identify fields and
chunks, mutating the contents of the formers while rearranging
the latters via addition, deletion, and splicing operations.

The two stages pick from a shared queue of inputs that
previous iterations of either stages incrementally built and
annotated. WEIZZ also maintains a global structure $CI$ to
keep track of comparison instructions possibly involved in
checksum tests when analyzing one or more inputs.

3.1. Surgical Stage

Algorithm 1 summarizes the working of the surgical
stage. Given an input $I$ picked from the queue, WEIZZ starts
by attempting to determine the dependencies between every
bit of the input taken individually and the comparison in-
structions in the program. The analysis is context-sensitive,
that is, when analyzing a comparison instruction we take
into account the calling context. We encode the latter using a
single word, and compute its exclusive OR with the address
of the instruction to identify the site of the comparison.

Procedure GETDEPS builds two data structures, both
indexed by a hash function ($Sites$ in Figure 2) for comparison
sites. A comparison table $CT$ stores values for operands
involved in observed instances (we keep as many as $|J|$) 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 informa-
tion. For each site, it iterates over each instance present in
$CT$ looking for input-to-state dependencies and checksum
information, and mutates bytes that can alter comparison
operands leveraging recorded values. While the third step
can run alongside the first two as portrayed in Figure 1, we
serialize them in Algorithm 1 for the sake of readability.

The stage then moves to tag construction, with each
input byte initially untagged. WEIZZ computes a topological
order over possibly found checksumming patterns: as we
will discuss later, this is useful when dealing with nested
checks. It then sorts comparison sites according to when
they were first encountered and processes them in this
order. Procedure PLACETAGS assigns tags to input bytes
taking into account input-to-state relationships $R$, checksum

3. Methodology

The fuzzing logic of WEIZZ comprises two stages. The
surgical one (top part of Figure 1) identifies dependencies
between an input and the comparison made in the program,
and restrict deterministic mutations to the sole bytes that
apparently influence operands of comparison instructions.
The analysis of dependencies can identify input-to-state
(use I2S for short in the following) relationships and
checksums, and places tags on input bytes to efficiently
summarize the discovered information for the next stage.

The structure-aware stage (bottom of Figure 1) extends
the nondeterministic working of AFL with structure-aware

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.

While developers have to write peach pits on purpose to
apply smart fuzzers to their software, in other cases writing a
format specification is a task already accomplished for other
purposes. LIBPROTOCOL-MUTATOR [31] provides support
for randomly mutating protocol buffers, which turns out
useful to generate seeds for classic grey-box fuzzers.

2.4. Discussion

Tables 1 depicts where our proposal 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.

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

Figure 1: Two-stage architecture of WEIZZ.

Table 1: Comparison with related approaches.
function SURGICALFUZZING(I):
1  CT, Deps ← GETDEPS(I)
2  foreach s ∈ Sites, j ∈ J do
3      R(s, j) ← DETECT12S(Deps, CT, s, j, I)
4  CI ← MARKCHECKSUMS(R(s, j), CI)
5  FUZZOPERANDs(I, CT, Deps, s, j)
6  Tags ← {0...|I|} with |Tags| = len(I)
7  o ← TOPOLOGICALSORT(CI, Deps)
8  foreach s: SITEsBYEXECORDER(CI, Deps) do
9      Tags ← PLACE_TAGS(Tags, s, CT, CI, R, o, I)
10     I, CI, fixOk ← FIXCHECKSUMs(CI, I, Tags, o)
11    CI ← PATCHALLCHECKSUMs(CI)
12  return I, Tags, fixOk

Algorithm 1: Surgical stage of WEIZZ.

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 b ∈ {0...7} do
5      CT' ← RUNINSTR(BITFLIP(b, b))
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] ∨
9        CT(s, j, op) ≠ CT'(s, j, op)
10  return CT, Deps

Algorithm 2: Dependency identification step.

information CI and the associated topological order o, the
initially computed dependencies Deps, and data associated
with comparison sites CT.

Like other fuzzers [5], [23] WEIZZ forces checksums by
patching involved instructions as it detects one, postponing
the input repairing process required to meet the unaltered
checksum condition. These are then carried out by procedure
FIXCHECKSUMs, which also discards false positives. At the
end of the surgical stage, WEIZZ re-applies patches to dodge
in later stages known checksums for unrepaired inputs.

In the next sections we describe each above-mentioned
step. The output of this 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.

3.1.1. 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 pro-
gram. As we discussed in Section 2.2, heavy-duty program
analyses providing means to this end is often costly, and
recent proposals like REDQUEEN and ECLIPSER explore
fast approximate ways to extract dependency information.
In this paper we go down this avenue by proposing a possibly
more accurate dependency identification technique.

Algorithm 2 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 also keeps a timestamp for
when RUNINSTR first observed it, and how many times it
encountered such a 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 as a
consequence of 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’ compared to CT.

Bit flips may however cause execution divergences, as
some comparisons are likely responsible for steering the
program towards an execution path or another1. 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 that we use to determine
whether execution did not diverge locally at the comparison
site. We prefer a local policy instead of enforcing that all
sites in CT and CT’ see the same number of instances
as we tolerate non-local divergences. 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.

As |J| is limited for efficiency, the reason why WEIZZ
tracks the most recent instead of the first encountered
instances of a comparison site is the following. Several fuzzers
apply trimming transformations to shorten inputs: there is a
good chance that such an input will explore a path hitting
a comparison site fewer than |J| times, and by choosing to
monitor the most recent instances we learn new facts when
analyzing longer inputs that lead to more than |J| instances.

3.1.2. Analysis of Comparison Site Instances. After con-
structing the dependencies, WEIZZ starts processing the data
recorded for up to |J| instances of each comparison site.

The first analysis is the detection of input-to-state cor-
respondences with DETECT12S. REDQUEEN explores this
capability 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 DETECT12S to every
operand in a comparison site instance and update the R data
structure with newly discovered I2S facts.

The second analysis involves the detection of checksum-
ing sequences with MARKCHECKSUMs. Similarly as in
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

1. When such an input is interesting (Section 2.1) we add it to the queue.
dependencies on some input bytes;
3) \( A_b(\text{Deps}(s, j, \text{op}_1)[b], \text{Deps}(s, j, \text{op}_2)[b]) = \text{false} \): 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, the \textsc{FuzzOperands} step replaces the deterministic mutations of AFL with surgical byte substitutions driven by data seen at comparison sites. For each input byte, \textsc{Weizz} determines each CT entry (i.e., each operand of a comparison site instance) it influences. Then \textsc{Weizz} replaces 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 that we add to the queue if its execution reveals a coverage improvement.

3.1.3. Tag Placement. \textsc{Weizz} succinctly describes dependency information between input bytes and performed comparisons by annotating such bytes with tags. Later steps like input repairing for checksums and structural information inference build heavily on the availability of such tags.

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

The tag assignment process relies on spatial and temporal locality in how comparison instructions get executed. \textsc{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 elaborate later, we extend this scheme to account for checksums and to reassign tags when heuristics spot a likely better candidate than the current choice.

Procedure \textsc{PlaceTags} (Algorithm 5 in the Appendix) iterates over comparison sites sorted by when first met in the execution, and attempts an inference for each input byte. We apply the procedure to instruction operands individually. If for the current byte \( b \) we find no dependency among all recorded instances, the cycle advances to the next byte (line 2), otherwise we compute a \( \text{numDeps} \) candidate, i.e., the number \( n \) of input bytes that affect the instruction (line 3). If the byte is untagged we tag it with the current instruction, otherwise we consider a reassignment. If the currently assigned tag \( \text{Tag}[b] \) does not come from a checksum check and \( n \) is smaller than \( \text{Tags}[b].\text{numDeps} \) (line 5), we reassign the tag as fewer dependencies suggest that the instruction may be more representative for the byte.

When \textsc{Weizz} finds a comparison treating the byte as from a checksum value (Section 3.1.2), we always assign it as its tag. To populate the \( \text{dependsOn} \) tag field we use a topological sort of the dependencies \( \text{Deps} \) over each input byte, that is, \textsc{Weizz} knows when a byte part of a checksum value represents also a byte that one or more outer checksums verify for integrity. \textsc{Weizz} will later repair inputs starting from the innermost checksum, with \( \text{dependsOn} \) pointing to the next comparison to process. Our technique is different than the one of \textsc{RedQueen}, which due to lack of accurate dependencies tries to build a topological sort based on clashes found in repair attempts with different orders.

3.1.4. Checksum Validation and Input Repair. The last conceptual step of the surgical stage involves checksum validation and input repair with respect to checksums patched in the program during previous iterations. In the process \textsc{Weizz} also discards false positives among such checksums.

Procedure \textsc{FixChecksums} (Algorithm 6 in the Appendix) filters tags marked as involved in checksum fields for which a patch is active, and processes them in the order described in the previous section. \textsc{Weizz} initially executes the patched program, obtaining a comparison table \( \text{CT} \) and the hash \( h \) of the coverage map as execution path footprint.

Then it iterates over each filtered tag as follows. It first extracts the input-derived operand value of the comparison instruction (i.e., the checksum computed by the program) and where in the input the contents of the other operand (which is I2S, see Section 3.1.2) are located, then it replaces the input bytes with the expected value (lines 4-5). \textsc{Weizz} makes a new instrumented execution over the repaired input, disabling the patch. If it handled the checksum correctly, the resulting hash \( h' \) will be identical to \( h \) implying no execution divergence happened, otherwise the checksum is a false positive and we have to bail.

After processing all the tags, we may still have patches left in the program for checksums not reflected by tags. \textsc{Weizz} may in fact erroneously treat a comparison instruction as a checksum due to inaccurate dependency information. \textsc{Weizz} removes all such patches and if after that the footprint for the execution does not change, the input repair process succeeded. This happens typically with patches for branch outcomes that turn out to be already “implied” by the repaired tag-dependent checksums.

Otherwise \textsc{Weizz} tries to determine which patch not reflected by tags may individually cause a divergence: it
tries patches one at a time and compares footprints to spot offending ones. This happens when WEIZZ initially finds a checksum for an input and the analysis of a different input reveals an overfitting, so we mark it as a false positive.

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. WEIZZ then also applies patches for checksums newly discovered by MARKCHECKSUMS in the current surgical stage, so when WEIZZ will analyze again the input (or similar ones from FUZZOPERAND) 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 assigns to the original input (Section 2.1). 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 [4], 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 inference strategy for fields and chunks is not sound, especially if compared to manually written format specifications, we give WEIZZ leeway in their 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 design 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, it allows WEIZZ to not commit to possibly erroneous choices when applying one of its heuristics, limiting their impact to the subsequent mutations, and granting more leeway to the overall fuzzing process in exploring alternate scenarios.

In the next sections we will describe how WEIZZ uses input tags to identify fields and chunks and the mutations it applies. 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 remaining ones 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

```plaintext
function FieldIdentification(input): as an example
int p = input[x];
if (p == magic) {
    return (chunk[4]==type[0] && chunk[5]==type[1] &&
           chunk[6]==type[2] && chunk[7]==type[3]);
}
```

in a field using a different instruction, as in the following fragment from the lodepng library:

```plaintext
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 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 FieldIdentification (Algorithm 3) 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 input-to-state, 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
function FINDCHUNKEND(Tags, k):
  end ← GoRightWhileSameTag(Tags, k)
  while Tags[end+1].ts >= Tags[k].ts do
    end ← FINDCHUNKEND(Tags, end+1)
  while Tags[end+1].id == Tags[k].parent do
    end ← end + 1
  end with probability Pr
end do
while Tags[end+1].id == Tags[k].ts do end ← end + 1
while Tags[end+1].ts >= Tags[k].ts do

Algorithm 4: Boundary identification for chunks.

struct ( // lines 2-3 )
  type;
  cksm;
  int x, y;
struct ( // 1-3 )
  type;
  cksm;
  int x, y;

int cksm // 4-5
int x, y // 2-3
int x, y // 8-9

Figure 4: Examples of chunks that WEIZZ can look for.

procedure FINDFIELDEND (Algorithm 7 in the Appendix), which looks for sequences of 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 relies on heuristics to locate plausible boundaries for chunks in the input sequence, then it then applies higher-order mutations to candidate chunks.

As a 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 PLACE_TAGS 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 4).

The procedure initially recognizes the 4 bytes of type (line 1), then it recursively looks for adjacent fields as long as 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 information. 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 FINDCHUNKEND, 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 copes better with this scenario. The heuristic 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 (Algorithm 8 in the Appendix).

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 information of the leading bytes in the current chunk. It picks one such chunk randomly and extends the current input by inserting the associated bytes before or after the current chunk. While AFLSMART relies on the specification for nesting information, we use the parent tag as a proxy for it.

- 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 of the current one (AFLSMART relies on type information), randomly picks one, and substitutes the bytes.
4. Implementation

We implemented our ideas on top of AFL 2.52b and QEMU 3.1.0 for x86-64 Linux targets. In the following we present instrumentation aspects specific to our approach, and discuss a few implementation tunings that we devised.

Instrumentation. WEIZZ computes context-sensitivity information for branch coverage and comparison tables by maintaining a shadow call stack in QEMU. To deal with the often large number of contexts in programs [32], like in ANGORA we extend the number of buckets in the coverage map from $2^{16}$ as in AFL to $2^{18}$, and compute the index using the source and destination basic block addresses and a one-word hash of the call stack. In the ANGORA experience, context-sensitive branch coverage may let a fuzzer explore programs more pervasively [9]. For CT we choose to host up to $2^{16}$ entries since there is one less degree of freedom in computing indexes in the Sites function (Figure 2).

Natural candidates for populating CT are cmp and sub instructions. However, like REDQUEEN we also record call instructions that may involve comparator functions: we check whether the locations for the first two arguments according to the Linux x86-64 ABI contain valid pointers, and dump 32 bytes for each operand\(^2\). 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.

WEIZZ may still miss dependencies between the input and the behavior of a program. This happens for instance in optimized code when a compiler uses arithmetic and logical operations to set CPU flags before a branch decision. In Section 6 we discuss possible remedies. We 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.

Deferred Surgical Fuzzing. Although the surgical stage provides crucial information on input dependencies, its execution can be quite expensive. For this reason, the current implementation of WEIZZ upon picking an input from the queue dynamically evaluates when it may be worth skipping it, advancing to the second stage. Similarly to AFLSMART, WEIZZ tracks the time elapsed since it last discovered an interesting input. The probability of entering the surgical stage increases with it and becomes one after 50 seconds.

However, an input that has no tag information can only undergo havoc mutations in the second stage. An important practical extension to our technique is the concept of derived tags. A derived tag is an educated guess on the actual tag based on tags seen in a similar input. Consider the PLACE_TAGS step in the surgical stage: when the assignment terminates, we can propagate this information to all the similar inputs that FUZZOPERANDS just added to the queue by processing the present input. Similarly, when we attempt high-order mutations over chunks we can: (i) for addition, copy the tags from other input to the newly added bytes; (ii) for deletion, keep the tags for the unaffected bytes; (iii) for splicing, use the tags from the other input. Derived tags in our experience are valuable to speed up the exploration, and eventually will be replaced by the actual tags when the input will undergo the surgical stage. In this scenario, the only case where we leave an input untagged is in presence of a havoc mutation that alters the input size.

5. Evaluation

We now present the results of a three-pronged evaluation of our prototype to address 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?

For RQ1 we analyze the code coverage from WEIZZ and a selection of fuzzers on applications evaluated in previous works (e.g., [4], [27], and [8]) and that process chunk-based formats. WEIZZ achieves high code coverage even when compared to AFLSMART that relies on a specification. For RQ2, we report on 11 bugs found in well-tested software, discussing 3 of them in more detail. Finally, for RQ3 we report on a few case studies suggesting that the tags we assign can strongly resemble the actual format, and that structural mutations are crucial for the effectiveness of WEIZZ.

**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. To compare different fuzzers, we measured the cumulative basic block coverage from running an application over the entire set of inputs generated by the fuzzer. We repeated each experiment 5 times, plotting the median value in the charts and reporting in the Appendix 60%-confidence intervals as in [5]. For the second stage of WEIZZ, we set $Pr_{field} = Pr_{chunk} = 1/15$, consistently with the probability of applying smart mutations in AFLSMART.

To facilitate reproducibility, we plan to release our experimental setup along with source code of WEIZZ.

5.1. RQ1: Chunk-based Formats

We evaluate the effectiveness in terms of code coverage for WEIZZ on the targets reported in Table 2. We start by comparing WEIZZ against the state-of-the-art solution for chunk-based formats AFLSMART, which can apply higher-order mutations over a virtual input structure (Section 2.3). We then extend our comparison considering general-purpose fuzzers that can still be quite effective in practice on this type of programs as seen in past works, e.g., [4] and [27].

5.1.1. AFLSMART. For this comparison we study the top 6 applications from Table 2 considered in past evaluations of AFLSMART to exercise 6 input formats. For AFLSMART
TABLE 2: Target applications considered in the evaluation.

| PROGRAM       | RELEASE     | INPUT FORMAT |
|---------------|-------------|--------------|
| wavpack       | 5.1.0       | WAV          |
| decompress    | 2.3.1-git   | JPEG         |
| ffmpeg        | 4.2.1       | AVI          |
| mpg42        | 2.3.16-27   | PNG          |
| readelf (GNU binutils) | 2.3.0       | ELF          |
| djpeg (libjpeg-turbo) | 1.5.6-6a68  | PNG          |
| oggdec (vorbis-tools) | 1.4.6       | MP3          |
| tcpdump       | 4.8.2       | PCAP         |
| gif2rgb (GIFLIB) | 5.2.2.1     | GIF          |

we use the peach pits written and released by its authors, and the latest release when we ran our tests (commit 604c40c).

We measured the code coverage achieved by: (a) AFLSMART with stacked mutations but without deterministic stages as suggested by its documentation, (b) WEIZZ in its standard configuration, and (c) a variant WEIZZ† where we disable Fuzz OPERANDS, checksum patching and input repairing. WEIZZ† allows us to focus on the effects of tag-based mutations, without benefits from our roadblock bypassing capabilities that AFLSMART misses. Nevertheless, 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 [33].

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 will 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. Hence, 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 the inference of structural facts may actually benefit WEIZZ†. The authors of AFLSMART acknowledge the importance of relaxed specification to expose imprecise implementations [4]: we will return to this in Section 6.

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 is crucial to generate valid inputs for programs that check data integrity, such as libpng that computes CRC-32 values over data.

Table 3 compares WEIZZ with the best alternative for code coverage at the end of the experiment. Interestingly, WEIZZ† is a better alternative than AFLSMART for 3 subjects. 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 turned out very effective to generate inputs that deviate significantly from the initial seed, e.g., mutating an AVI file into an MPEG-4 one. In Section 5.2 we will back this claim with a few case studies.

5.1.2. Grey-box Fuzzers. For this experiment we analyze the bottom 8 programs from Table 2. These are popular open source applications handling chunk-based formats that the community has heavily tested (e.g., by the OSS-Fuzz project for continuous fuzzing [34]), and were used in past evaluations of state-of-the-art fuzzers like [4], [5], [8], [19].

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 instru-

TABLE 3: WEIZZ vs. best alternative for targets of Figure 5: a basic block (BB) coverage percentage higher than 100 means that WEIZZ outperforms the best alternative fuzzer.

| PROGRAM       | BEST ALTERNATIVE | BB % W.R.T. BEST | CRASHES FOUND BY |
|---------------|------------------|------------------|------------------|
| wavpack       | AFLSMART         | 106%             | all              |
| readelf       | WEIZZ†           | 110%             | none             |
| decompress    | AFLSMART         | 100%             | none             |
| djpeg         | AFLSMART         | 96%              | none             |
| libpng        | WEIZZ†           | 132%             | none             |
| ffmpeg        | WEIZZ†           | 114%             | WEIZZ            |

Figure 5: Basic block coverage over time (5 hours).
TABLE 4: WEIZZ vs. best alternative for targets of Figure 6. Columns have the same meaning used in Table 3.

| PROGRAM | BEST ALTERNATIVE | BB % w.r.t. Best | CRASHES FOUND BY |
|---------|------------------|------------------|------------------|
| djpeg   | AFL++            | 105%             | none             |
| libpng  | AFL++            | 120%             | none             |
| mpg321  | AFL++            | 100%             | WEIZZ, AFL++     |
| oggdec  | ECLIPSER         | 108%             | none             |
| readelf | AFL++            | 175%             | none             |
| tcpdump | ECLIPSER         | 117%             | none             |
| gif2rgb | AFL+++           | 100%             | none             |

Figure 6 plots the median basic block coverage over time. Consistently with previous evaluations [27], 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 input1 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 in the list above process formats with checksums that only WEIZZ repairs, although ECLIPSER appears to catch up over time for oggdec. The structural mutation capabilities combined with this factor may explain the gap between WEIZZ and other fuzzers. For objdump and readelf, by the way REDQUEEN outperforms other fuzzers over them in its evaluation [5] we may speculate I2S facts3 are boosting the work of WEIZZ.

On mpg321 and gif2rgb, AFL-based fuzzers perform very similarly, with a large margin over ECLIPSER, confirming that the AFL standard mutations can be effective on some chunk-based formats. Finally, WEIZZ leads for djpeg but the other fuzzers are not too far from it.

Table 4 compares WEIZZ with the best alternative when the budget expires. Interestingly, AFL is the best alternative for 3 programs. 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: Zero-day Bugs

A crucial feature of WEIZZ is that it does not require a format specification for a program, allowing users to analyze quite different applications without additional effort.

To explore the effectiveness of WEIZZ, we ran it on several real-world targets, including software to process inputs not strictly falling within the chunk-based paradigm. WEIZZ revealed 11 zero-day bugs in 7 popular, well-tested applications: 6 bugs are NULL pointer dereferences (CWE-476), 1 involves an unaligned realloc (CWE-761), 2 can lead to buffer overflows (CWE-122), 1 causes an out-of-bounds read (CWE-125), and 1 a division by zero (CWE-369). Table 5 reports all the bugs. Crashes from the RQ1 experiments that involve known bugs do not appear in it. We now present details on three bugs that can provide interesting insights on the effectiveness and versatility of WEIZZ.

CUPS. The Common UNIX Printing System is an open source project developed by Apple and used in many UNIX-like systems. While its HTML interface is not chunk-oriented, we checked whether WEIZZ could mutate the HTTP requests enclosing it. Interestingly, WEIZZ crafted a request that led CUPS to reallocate a user-controlled buffer, paving the road to a House of Spirit attack [35]. The key to finding the bug was to have FUZZOPERANDS replace some

1. The not_kitty.png AFL test case for libpng, and a PCAP of a few seconds from a capture-the-flag competition for tcpdump.
2. For objdump logging function call arguments turns out crucial, see [5].
TABLE 5: Bugs found and reported in real-world software.

| Program       | Bug ID   | Type |
|---------------|----------|------|
| objdump       | Bugzilla #24938 | CWE-476 |
| CUPS          | rdar://problem/5000749 | CWE-761 |
| libmirage (CDEmu) | CVE-2019-15540 | CWE-122 |
| libmirage (CDEmu) | CVE-2019-15757 | CWE-476 |
| dimg2img      | Launchpad #1835461 | CWE-761 |
| dimg2img      | Launchpad #1835463 | CWE-122 |
| jpegEnc       | GitHub #65 | CWE-476 |
| mpg/321       | Launchpad #1842445 | CWE-122 |
| libavformat (ffmpeg) | Ticket #8335 | CWE-369 |

current input bytes with I2S facts (‘Accept-Language’ logged as operand for a call to _cups_strcasecmp), thus materializing a valid request field that chunk mutations later duplicated. Apple confirmed the bug and fixed it in the release 2.3b8. We also reported a NULL pointer dereference from an HTTP request, fixed in the release 2.3.0.

libMirage. This library provides uniform access to data stored in several disk image formats. Weizz identified a critical bug: an attacker can generate a heap-based buffer overflow, which in turn may corrupt allocator metadata and even grant root access to the attacker, as the CDEmu daemon using it runs with root privileges on most Linux systems. Interestingly, we fed Weizz with an ISO image as initial seed and it exposed the bug in the NRG image parser, demonstrating how it can deviate even considerably among input formats based on how the code handles input bytes.

objdump. The GNU binary utilities (binutils) are one of the most tested program suites (e.g., [34]). Weizz found a NULL pointer dereference inside the objdump tool for displaying information about object files. Similary as with libMirage, Weizz exposed a bug in code for the PE executable format, while we used an ELF binary as initial seed.

5.3. RQ3: Understanding the Impact of Tags

A fascinating question is why Weizz is effective over the programs we considered. Pinpointing the root causes in the general case is far from trivial, since targets may follow very different approaches to processing input bytes, and involve heterogeneous formats. We partially answer this question with two case studies: (i) we explore to which extent tags found by Weizz on one seed for libpng can assist field and chunk identification, and (ii) we show that both tag-based mutations and roadblock bypassing are essential for ffmpeg, but either alone is not enough for optimal coverage.

Tag Accuracy. Figure 7 shows the raw bytes from the initial seed not_kitty.png for libpng. The test case starts with a 8-byte magic number (in orange) and then contains 5 chunks. Each chunk is characterized by four fields: length (4 bytes, in green), type (4 bytes, red), data (typically heterogeneous, as with PNG_CHUNK_IHDR data in yellow), and a CRC-32 checksum (4 bytes, in blue). The last chunk has no data since it merely marks the file end.

To test 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 field we found in square brackets, and underline bytes that Weizz deems from checksum fields. Weizz identifies correct boundaries with a few exceptions. The last three fields in the file are not revealed, as they are part of a chunk that libpng never accesses. In some cases a data field is merged with the adjacent checksum value. The reason is that libpng does not execute comparisons over the data bytes but first computes their checksum, and verifies it against the expected value using a comparison that will characterize also the data bytes (for the data-flow dependency). Dependency analysis on the operands (Section 5.1.2) however identifies the checksum correctly for FIXCHECKSUMS, and when the input gets repaired and eventually enters the data field stage again, libpng will execute new comparison instructions over the data bytes, and Weizz will correctly tag them with such instructions. Finally, the IDAT DATA parts in grey can be seen as a binary blob fragmented by Weizz into distinct fields with gaps due to untagged bytes, since not all the routines for their processing perform comparisons over the blob bytes.

The analysis of chunk boundaries is more challenging, as FINDFIELDEND is sensitive to the initial position for the analysis. For instance, when starting from the 8-byte magic number it correctly identifies the entire test case as one chunk. Similarly, when starting from a byte of a length field, it may capture all the other fields for that chunk. However, when starting from a type field it may build an incomplete chunk. Nonetheless, even when boundary detection is imprecise, 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 can sometimes be beneficial in order to exercise not so well-tested corner cases or shared parsing code: we will return to this point in Section 6. For libpng Weizz achieved better coverage than other fuzzers, including AFLSMART.

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? Let us consider ffmpeg as case 5. Depending on the starting byte, it could miss some of the first bytes.
We show the coverage from AFLSMART for reference, but not the queue size as its context-insensitivity brings a different queue handling.

considered. We leave to future work the investigation of using forms of bit-level DTA \cite{[36] \cite{[37] as an alternative for dependency identification over “costly” inputs.

In the implementation, WEIZZ misses dependencies for comparisons made with instructions other than the currently supported ones (e.g., \texttt{lea}, \texttt{add}), as logging them blindly can be expensive. In the spirit of \textsc{VUZZER}, we could resort to an intra-procedural static analysis to point out those actually involved in branching decisions. Similarly, we could use heuristics to reveal code targets for switch jump tables. Like \textsc{REDQUEEN}, WEIZZ can detect but not patch checksums tests made via comparison functions, yet we could extend it to hook library functions of this kind (e.g., \texttt{memcmp}).

The structure-aware stage offers automatic means to effectively fuzz chunk-based formats. In addition to not requiring a specification, our approach 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 that we pointed out in Section 5 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 \cite{[4]}. We find this consistent with the \textsc{GRIMOIRE} experience with grammars, where paths outside the specification revealed coding mistakes \cite{[6]}, as we discussed also in Section 1.

Yet there is room to extend the chunk boundary inference schemes with new heuristics, or make the current ones more efficient. For instance, we have explored 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. For the current heuristics, we may refine the probability of selecting the initial byte by weighing in changes in timestamp monotonicity that impact \textsc{FINDCHUNKEND}. We leave these extensions to future work.

6. Discussion

WeIZZ introduces novel ideas in the fuzzing landscape for computing dependency information between input bytes and comparisons in the code in order to simultaneously overcome fuzzing roadblocks and back a fully automatic analysis of chunk-based binary formats. From a practical standpoint, the approach has two advantages: (i) fuzzers already attempt bitflips during deterministic mutations, and (ii) instrumenting comparisons is becoming a common practice to overcome roadblocks. We empower such analyses to better characterize the program behavior while fuzzing, paving the way to the tag assignment mechanism. Prior proposals do not offer sufficient information to this end: even in the case of \textsc{REDQUEEN}, its colorization scheme \cite{[5]} identifies I2S portions of an input (crucial for roadblocks), but cannot reveal dependencies for non-I2S bytes.

A downside of our technique is that analyzing dependencies via flipping can get costly over large inputs. However, an equally important factor is the time the program takes to execute one test case. In our experiments WEIZZ applied the surgical stage to inputs long up to 3K bytes, achieving comparable or better coverage than the other fuzzers we

Figure 8: Five-hour analysis of ffmpeg: WEIZZ\textsuperscript{1} lacks roadblock bypass techniques, WEIZZ\textsuperscript{2} lacks tag-based mutations.
A. Appendix

A.1 Roadblocks from lodepng

As an example of roadblocks found in real-world software, the following excerpt from lodepng image encoder and decoder looks for the presence of the magic string IHDR denoting the namesake PNG chunk type and checks the integrity of its data against a CRC-32 checksum:

```c
if (!lodepng_chunk_type_equals(in + 8, "IHDR")) {  
  /error: it doesn't start with a IHDR chunk*/  
  CERROR_RETURN_ERROR(state->error, 29);
}
unsigned CRC = lodepng_read32bitInt(&in[29]);
unsigned checksum = lodepng_crc32(&in[12], 17);
if (CRC != checksum) {  
  /invalid CRC*/  
  CERROR_RETURN_ERROR(state->error, 57);
}
```

A.2 Additional Algorithms

Tag Placement. Algorithm 5 provides pseudocode for the PLACETAGS procedure, while we refer the reader to Section 3.1.3 for a discussion of its main steps. An interesting detail presented here is related to the condition \( n' > 4 \) when computing the reassign flag. This results from experimental observations, with 4 being the most common field size in the formats we analyzed. Hence, WEIZZ will choose as characterizing instruction for a byte the first comparison that depends on no more than 4 bytes, ruling out possibly “shorter” later comparisons as they may be accidental.

Checksum Handling. In Algorithm 6 we report detailed pseudocode for the FIXCHECKSUMS procedure that we illustrated in its main steps throughout Section 3.1.4.

Field Identification. Algorithm 7 shows how to implement the FINDFIELDEND procedure that we use to detect fields complying to one of the patterns from Figure 3, which we discussed in Section 3.2.1. To limit the recursion at line 3 for field extension, we use 8 as maximum depth (line 7).

Chunk Identification. As discussed in Section 3.2.2, 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. \textsc{Weizz} chooses between the original and the alternative heuristic with a probability $\Pr_{\text{chunk}}$ as described in Algorithm 8.

### A.3 Implementation Tuning

In the following we present present additional optimizations that we integrated in our implementation.

**Loop Bucketization.** In addition to newly encountered edges, AFL deems a path interesting when some branch hit count changes (Section 2.1). To avoid keeping too many inputs, AFL verifies whether the counter falls in a previously unseen power-of-two interval. However, as pointed out by the authors of LAF-INTEL [14], this approach falls short in deeming progress with patterns like the following:

```c
for (int i = 0; i < strlen(magic); ++i)
  if (magic[i] != input[i]) return;
```

To mitigate this problem, in the surgical phase \textsc{Weizz} considers interesting also inputs that lead to the largest hit count observed for an edge across the entire stage.

**Optimizing I2S Fuzzing.** Another tuning for the surgical stage involves the FUZZOPERANDS step. \textsc{Weizz} decides to skip a comparison site when previous iterations over it were unable to generate any coverage-improving paths. In particular, the probability of executing FUZZOPERANDS on a site is initially one and decreases with the ratio between failed and total attempts.

### A.4 Additional Experimental Data

Table 6 and Table 7 provide the 60%-confidence intervals for the experiments reporting the basic block code coverage discussed in Section 5.1.

**TABLE 6: Basic block coverage (60% confidence intervals) after 5-hour fuzzing of the top 6 subjects of Table 2.**

| PROGRAMS | \textsc{Weizz} | \textsc{AFLSmart} | \textsc{Weizz}† |
|----------|---------------|-----------------|---------------|
| wavpack  | 1824-1887     | 1738-1813       | 1614-1749     |
| readelf  | 7298-7370     | 6087-6188       | 6586-6731     |
| decompress | 5831-6276     | 6027-6569       | 5376-5685     |
| djpeg    | 2109-2137     | 2214-2221       | 2121-2169     |
| libpng  | 1620-1688     | 1000-1035       | 1188-1231     |
|ffmpeg    | 15946-17885   | 9352-9923       | 14515-14885   |

**TABLE 7: Basic block coverage (60% confidence intervals) after 24-hour fuzzing of the bottom 8 subjects of Table 2.**

| PROGRAMS | \textsc{Weizz} | \textsc{Eclipsizer} | \textsc{AFL++} | \textsc{AFL} |
|----------|---------------|---------------------|---------------|-------------|
| djpeg    | 612-614       | 492-532             | 561-577       | 581-592     |
| libpng  | 1747-1804     | 704-711             | 877-901       | 987-989     |
| objdump  | 3366-4235     | 2549-2648           | 2756-3748     | 2451-2723   |
| mpg321   | 428-451       | 204-204             | 426-427       | 204-204     |
| oggdec   | 369-372       | 332-346             | 236-244       | 211-211     |
| readelf  | 7428-7603     | 2542-2871           | 4265-5424     | 2982-3091   |
| tcpdump  | 7662-7833     | 6591-6720           | 5033-5453     | 4471-4576   |
| gif2rgb  | 453-464       | 357-407             | 451-454       | 457-465     |