Smart Greybox Fuzzing

Van-Thuan Pham*   Marcel Böhme†   Andrew E. Santosa*
Alexandru Răzvan Căciulescu‡   Abhik Roychoudhury*

*National University of Singapore, Singapore   thuanpv.nus@gmail.com, santosa_1999@yahoo.com
abhik@comp.nus.edu.sg
†Monash University, Australia   marcel.boehme@acm.org
‡University Politehnica of Bucharest, Romania   alexandru.razvan.c@gmail.com

Abstract—**Coverage-based greybox fuzzing (CGF) is one of the most successful methods for automated vulnerability detection. Given a seed file (as a sequence of bits), CGF randomly flips, deletes or bits to generate new files. CGF iteratively constructs (and fuzzes) a seed corpus by retaining those generated files which enhance coverage. However, random bitflips are unlikely to produce valid files (or valid chunks in files), for applications processing complex file formats.**

In this work, we introduce smart greybox fuzzing (SGF) which leverages a high-level structural representation of the seed file to generate new files. We define innovative mutation operators that work on the virtual file structure rather than on the bit level which allows SGF to explore completely new input domains while maintaining file validity. We introduce a novel validity-based power schedule that enables SGF to spend more time generating files that are more likely to pass the parsing stage of the program, which can expose vulnerabilities much deeper in the processing logic.

Our evaluation demonstrates the effectiveness of SGF. On several libraries that parse structurally complex files, our tool AFLSMART explores substantially more paths (up to 200%) and exposes more vulnerabilities than baseline AFL. Our tool AFLSMART has discovered 42 zero-day vulnerabilities in widely-used, well-tested tools and libraries; so far 17 CVEs were assigned.

1. Introduction

Coverage-based greybox fuzzing (CGF) is a popular and effective approach for software vulnerability detection. As opposed to blackbox approaches which suffer from a lack of knowledge about the application, and whitebox approaches which incur high overheads due to program analysis and constraint solving, greybox approaches use lightweight code instrumentation. The American Fuzzy Lop (AFL) fuzzer [31] and its extensions [1], [2], [7], [17], [18], [21], [27] constitute the most widely-used embodiment of CGF.

CGF technology proceeds by input space exploration via mutation. Starting with seed inputs, it mutates them using a pre-defined set of generic mutation operators (such as bit-flips). Control flows exercised by the mutated inputs are then examined to determine whether they are sufficiently "interesting". The lightweight program instrumentation helps the fuzzer make this judgment on the novelty of the control flows. Subsequently, the mutated inputs which are deemed sufficiently new are submitted for further investigation, at which point they are mutated further to explore more inputs. The aim is to enhance greater behavioral coverage, and to expose more vulnerabilities in a limited time budget.

One of the most significant and well-known limitations of CGF is its *lack of input structure awareness*. The mutation operators of CGF work on the *bit-level representation* of the seed file. Random bits are flipped, deleted, added, or copied from the same or from a different seed file. Yet, many security-critical applications and libraries will process highly structured inputs, such as image, audio, video, database, document, or spreadsheet files. Finding vulnerabilities effectively in applications processing such widely used formats is of imminent need. Mutations of the bit-level file representation are unlikely to effect any structural changes on the file that are necessary to effectively explore the vast yet sparse domain of valid program inputs. More likely than not arbitrary bit-level mutations of a valid file will result in an invalid file that is rejected by the program's parser before reaching the data processing portion of the program.

To tackle this problem, two main approaches have been proposed that are based on dictionaries [30] and dynamic taint analysis [25]. Michał Zalewski, the creator of AFL, introduced the *dictionary*, a lightweight technique to inject interesting byte sequences or tokens into the seed file during mutation at random locations. Zalewski's main concern [35] was that a full support of input awareness might come at a cost of efficiency or usability, both of which are AFL's secret to success. AFL benefits tremendously from a dictionary when it needs to come up with magic numbers or chunk identifiers to explore new paths. Rawat et al. [25] leverage dynamic taint analysis [26] and control flow analysis to infer the locations and the types of the input data based on which their tool (VUZZER) knows where and how to mutate the input effectively. However, both the dictionary and taint-based approaches do not solve our primary problem: to mutate the high-level structural representation of the file rather than its bit-level representation. For instance, neither a dictionary nor an inferred program feature help in adding or deleting complete chunks from a file.

In contrast to CGF, smart blackbox fuzzers [15], [38] are already input-structure aware and leverage a model of the file format to construct new valid files from existing valid files. For instance, Peach [38] uses an input model

to disassemble valid files and to reassemble them to new valid files, to delete chunks, and to modify important data values. LangFuzz [15] leverages a context-free grammar for JavaScript (JS) to extract code fragments from JS files and to reassemble them to new JS files. However, awareness of input structure alone is insufficient and the coverage-feedback of a greybox fuzzer is urgently needed – as shown by our experiments with Peach. In our experiments Peach performs much worse even than AFL, our baseline greybox fuzzer. Our detailed investigation revealed that Peach does not reuse the generated inputs that improve coverage for further test input generation. For instance, if Peach generated a WAV-file with a different (interesting) number of channels, that file could not be used to generate further WAV-files with the newly discovered program behaviour. Without coverage-feedback interesting files will not be retained for further fuzzing. On the other hand, retaining all generated files would hardly be economical.

In this paper, we introduce *smart greybox fuzzing* (SGF)—which leverages a high-level structural representation of the seed file to generate new files—and investigate the impact on fuzzer efficiency and usability. We define innovative mutation operators that work on the virtual structure of the file rather than on the bit level. These *structural mutation operators* allow SGF to explore completely new input domains while maintaining the validity of the generated files. We introduce a novel *validity-based power schedule* that assigns more energy to seeds with a higher degree of validity and enables SGF to spend more time generating files that are more likely to pass the parsing stage of the program to discover vulnerabilities deep in the processing logic of the program.

We implement AFLSMART, a robust yet efficient and easy-to-use *smart* greybox fuzzer based on AFL, a popular and very successful CGF. AFLSMART integrates the input-structure component of Peach with the coverage-feedback component of AFL. Hence, in our evaluation we compare against both as baseline techniques. Our evaluation demonstrates that AFLSMART, within a given time limit of 24 hours, can double the zero-day bugs found. AFLSMART discovers 33 bugs (8 CVEs assigned) while the baseline (AFL and its extension AFLFAST [2]) can detect only 16 bugs, in large, widely-used, and well-fuzzed open-source software projects, such as FFmpeg, LibAV, LibPNG, Wavpack, OpenJPEG and Binutils. AFLSMART also significantly improves the path coverage up to 200% compared to the baseline. AFLSMART also outperforms VUZZER [25] on its benchmarks; AFLSMART discovers seven (7) bugs which VUZZER could not find in another set of popular open-source programs, such as *tcpdump*, *tcptrace* and *gif2png*. Moreover, in a 1-week bug hunting campaign for FFmpeg, AFLSMART discovers nine (9) more zero-day bugs (9 CVEs assigned). Its effectiveness comes with negligible overhead – with our optimization of *deferred cracking* AFLSMART achieves execution speeds which are similar to AFL.

In our experience with AFLSMART, the time spent writing a file format specification is outweighed by the tremendous improvement in behavioral coverage and the number of bugs exposed. One of us spent five working days to develop 10 file format specifications (as Peach Pits [38]) which were used to fuzz all 16 subject programs. Hence, once developed, file format specifications can be reused across programs as well as for different versions of the same program.

In summary, the main contribution of our work is to make greybox fuzzing input format-aware. Given an input format specification (e.g., a Peach Pit [38]), our *smart greybox fuzzer* derives a structural representation of the seed file, called virtual structure, and leverages our novel smart mutation operators to modify the virtual file structure in addition to the file's bit sequence during the generation of new input files. We propose smart mutation operators, which are likely to preserve the satisfaction w.r.t. a file format specification. During the greybox fuzzing search, our tool AFLSMART measures the degree of validity of the inputs produced with respect to the file format specification. It prioritizes valid inputs over invalid ones, by enabling the fuzzer to explore more mutations of a valid file as opposed to an invalid one. As a result, our smart fuzzer largely explores the restricted space of inputs which are valid as per the file format specification, and attempts to locate vulnerabilities in the file processing logic by running inputs in this restricted space. We conduct extensive evaluation on well-tested subjects processing complex file formats such as PNG and WAV. Our experiments demonstrate that the smart mutation operators and the validity-based power schedule introduced by us, increases the effectiveness of fuzzing both in terms of path coverage and vulnerabilities found within a time limit of 24 hours. These results also demonstrate that the additional effectiveness in our smart fuzzer AFLSMART is *not* achieved by sacrificing the efficiency of greybox fuzzing and AFL.

2. Motivating Example

2.1. The WAVE File Format

Most file systems store information as a long string of zeros and ones—a file. It is the task of the program to make sense of this sequence of bits, i.e., to parse the file, and to extract the relevant information. This information is often structured in a hierarchical manner which requires the file to contain additional structural information. The structure of files of the same type is defined in a file format. Adherence to the file format allows the same file to be processed by different programs.

WAVE files (`*.wav`) contain audio information and can be processed by various media players and editors. A WAVE file consists of *chunks* (see Figure 1). Each chunk consists of chunk identifier, chunk length and chunk data. Chunks are structured in a hierarchical manner. The root chunk requires the first four bytes of the file to spell (in unicode) RIFF followed by four bytes specifying the total size n of the children chunks plus four. The next four bytes must spell (in unicode) WAVE. The remainder of a WAVE file contains the

Chunk Type	Field	Length	Contents
RIFF	ckID	4	Chunk ID: RIFF
	cksize	4	Chunk size: $4+n$
	WAVEID	4	WAVE id: WAVE
	chunks	n	Chunks containing format information and sampled data
fmt	ckID	4	Chunk ID: fmt
	cksize	4	Chunk size: 16, 18 or 40
	wFormatTag	2	Format code
	nChannels	2	Number of interleaved channels
	nSamplesPerSec	4	Sampling rate (blocks per second)
	...		
Optional chunks (fact chunk, cue chunk, playlist chunk, ...)			
data	ckID	4	Chunk ID: data
	cksize	4	Chunk size: n
	sampled data	n	Samples
	pad byte	0 or 1	Padding byte if n is odd

Figure 1: An excerpt of the WAVE file format (from Ref. [34])

children chunks, the mandatory fmt chunk, several optional chunks, and the data chunk. The data chunk itself is subject to further structural constraints.

We can clearly see that a WAVE file embeds audio information and meta-data in a hierarchical chunk structure. The WAVE file format governs all WAVE files and allows for efficient and systematic parsing of the audio information.

2.2. The Anatomy of a Vulnerability in a Popular Audio Compression Library

In the following, we discuss a vulnerability that our smart greybox fuzzer AFLSMART found in WavPack [40], a popular audio compression library that is used by many well-known media players and editors such as Winamp, VLC Media Player, and Adobe Audition. In our experiments, the same vulnerability could not be found by traditional greybox fuzzers such as AFL [31] or AFLFAST [2].

The discovered vulnerability (CVE-2018-10536) is a *buffer overwrite* in the WAVE-parser component of Wav-Pack. To construct an exploit, a *WAVE file* with more than one format chunks needs to be crafted that satisfies several complex structural conditions. The WAVE file contains the mandatory riff, fmt, and data chunks, plus an additional fmt chunk placed right after the first *fmt* chunk. The first fmt chunk specifies IEEE 754 32-bits (single-precision) floating point (*IEEE float*) as the waveform data format (i.e., fmt.wFormatTag= 3) and passes all sanity checks. The second fmt chunk specifies PCM as the waveform data format, one channel, one bit per sample, and one block align (i.e., fmt.wFormatTag= 1, fmt.nChannels= 1, fmt.nBlockAlign=1, and fmt.wBitsPerSample= 1).

The first fmt chunk configures WavPack to read the data in *IEEE float* format, which requires certain constraints to be satisfied, e.g., on the number of bits per sample (Lines

```
1  else if (!strncmp (chunk_header.ckID, "fmt ", 4)){
2    DoReadFile (infile, &WaveHeader, ...)
3    format = WaveHeader.FormatTag;
4    config->bits_per_sample = WaveHeader.BitsPerSample;
5    // Sanity checks
6    if (format == 3 && config->bits_per_sample != 32)
7      supported = FALSE;
8    if (WaveHeader.BlockAlign / WaveHeader.NumChannels
         < (config->bits_per_sample + 7) / 8)
9      supported = FALSE;
10   if (!supported) exit();
11   if (format==3) config->float_norm_exp=CONFIG_FLOAT;
12   ...
```

Figure 2: Sketching cli/riff.c @ revision 0a72951

6–10). The second fmt chunk allows to override certain values, e.g., the number of bits per sample, while maintaining the *IEEE float* format configuration. More specifically, the fmt-handling code is shown in Figure 2. The first fmt chunk is parsed as format 3 (IEEE float), 32 bits per sample, 1 channel, and 4 block align (Lines 2–4). The configuration passes all sanity checks for an *IEEE float* format (Lines 6–10), and sets the global configuration accordingly (Line 11). The second fmt chunk is parsed as format 1 (PCM), 1 bits per sample, 1 channel, and 1 block align (Lines 2–4). The new configuration would be valid if WavPack had not maintained *IEEE float* as the waveform data and had reset float_norm_exp. However, it does maintain *IEEE float* and thus allows an invalid configuration that would otherwise not pass the sanity checks which finally leads to a buffer overwrite that can be controlled by the attacker.

The vulnerability was patched by aborting when the *.wav file contains more than one fmt chunk. A similar vulnerability (CVE-2018-10537) was discovered and patched for *.w64 (WAVE64) files.

2.3. Difficulties of Traditional Greybox Fuzzing

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Corpus S
1: **repeat**
2:    $s = $ CHOOSENEXT(S) *// Search Strategy*
3:    $p = $ ASSIGNENERGY(s) *// Power Schedule*
4:    **for** i from 1 to p **do**
5:       $s' = $ MUTATE_INPUT(s)
6:       **if** s' crashes **then**
7:          add s' to $S_{\text{✗}}$
8:       **else if** ISINTERESTING(s') **then**
9:          add s' to S
10:      **end if**
11:    **end for**
12: **until** *timeout* reached or *abort*-signal
Output: Crashing Inputs $S_{\text{✗}}$

We use these vulnerabilities to illustrate the shortcomings of traditional greybox fuzzing. Algorithm 1, which is extracted from [2], shows the general greybox fuzzing loop. The fuzzer is provided with a initial set of program inputs, called *seed corpus*. In our example, this could be a set of WAVE files that we know to be valid. The greybox fuzzer mutates these seed inputs in a continuous loop to generate new inputs. Any new input that increases the coverage is added to the seed corpus. A well-known and very successful coverage-based greybox fuzzer is American Fuzzy Lop (AFL) [31].

Guidance. A coverage-based greybox fuzzer is guided by a search strategy and a power schedule. The *search strategy* decides the order in which seeds are chosen from the seed corpus, and is implemented in CHOOSENEXT (Line 2). The *power schedule* decides a seed's energy, i.e., how many inputs are generated by fuzzing the seed, and is implemented in ASSIGNENERGY (Line 3). For instance, AFL spends more energy fuzzing seeds that are small and execute quickly.

Bit-level mutation. Traditional greybox fuzzers are unaware of the input structure. In order to generate new inputs, a seed is modified according to pre-defined mutation operators. A *mutation operator* is a transformation rule. For instance, a bit-flip operator turns a zero into a one, and vice versa. Given a seed input, a mutation site is randomly chosen in the seed input and a mutation operator applied to generate a new test input. In Algorithm 1, the method MUTATE_INPUT implements the input generation by seed mutation. These mutation operators are specified on the *bit-level*. For instance, AFL has several deletion operators, all of which delete a contiguous, fixed-length sequence of bits in the seed file. AFL also has several addition operators, for instance to add a sequence of only zero's or one's, a random sequence of bits, or to copy a sequence of bits within the file. For our motivating example, Figure 3 shows the first 72 bytes of a canonical WAVE file. To expose CVE-2018-10536, a second *valid* `fmt` chunk must be added in-between the existing `fmt` and `data` chunks. Clearly, it is extremely unlikely for AFL to apply a sequence of bit-level mutation operators to the file that result in the insertion of such additional, valid chunks.

Dictionary. To better facilitate the fuzzing of structured files, many greybox fuzzers, including AFL, allow to specify a list of interesting byte sequences, called dictionary. In our motivating example, such byte sequences could be words, such as `RIFF`, `fmt`, and `data` in unicode, or common values, such as 22050 and 88200 in hexadecimal. However, a dictionary will not contribute much to the complex task of constructing a valid chunk that is inserted right at the joint boundary of two other chunks.

3. Smart Greybox Fuzzing

Smart greybox fuzzing (SGF) is more effective than both, smart blackbox fuzzing and traditional greybox fuzzing. Unlike traditional greybox fuzzing, SGF allows to penetrate deeply into a program that takes highly-structured inputs without getting stuck in the program's parser code.

Stored Bits	Information		Description
52 49 46 46	R I F F		RIFF.`ckID`
24 08 00 00		2084	RIFF.`cksize`
57 41 56 45	W A V E		RIFF.`WAVEID`
66 6d 74 20	f m t ␣		fmt.`ckID`
10 00 00 00		16	fmt.`cksize`
01 00 02 00	1	2	fmt.`wFormatTag` (1=PCM) & fmt.`nChannels`
22 56 00 00		22050	fmt.`nSamplesPerSec`
88 58 01 00		88200	fmt.`nAvgBytesPerSec`
04 00 10 00	4	16	fmt.`nBlockAlign` & fmt.`wBitsPerSample`
64 61 74 61	d a t a		data.`ckID`
00 08 00 00		2048	data.`cksize`
00 00 00 00	sound data 1		left and right channel
24 17 1e f3	sound data 2		left and right channel
3c 13 3c 14	sound data 3		left and right channel
16 f9 18 f9	sound data 4		left and right channel
34 e7 23 a6	sound data 5		left and right channel
3c f2 24 f2	sound data 6		left and right channel
11 ce 1a 0d	sound data 7		left and right channel
...			

Figure 3: Canonical WAVE file (from Ref. [34])

Unlike smart blackbox fuzzing, SGF leverages coverage-information to explore the program's behavior more efficiently.

3.1. Virtual Structure

The effectiveness of SGF comes from the careful design of its smart mutation operators. First, these operators should fully leverage the structural information extracted from the seed inputs to apply higher-order manipulations at both the chunk level and the bit level. Second, they should be unified operators to support all chunk-based file formats (e.g., MP3, ELF, PNG, JPEG, WAV, AVI, PCAP). Last but not the least, all these operators must be lightweight so that we can retain the efficiency of greybox fuzzing.
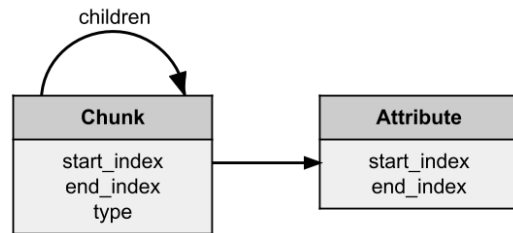

Figure 4: Virtual structure used by AFLSMART

To implement these three design principles, we introduce a new lightweight yet generic data structure namely *virtual structure* which can facilitate the structural mutation operators. Each input file can be represented as a (parse) tree. The nodes of this tree are called chunks or attributes, with the *chunks* being the internal nodes of the tree and the *attributes* being the leaf nodes of the tree.

A *chunk* is a contiguous sequence of bytes in the file. There is a *root chunk* spanning the entire file. As visualized

in Fig. 4, each chunk has a *start- and an end-index* representing the start and end of the byte sequence in the file, and a *type* representing the distinction to other chunks (e.g., an `fmt` chunk is different from a `data` chunk in the WAVE file format). Each chunk can have zero or more chunks as *children* and zero or more attributes. An *attribute* represents important data in the file that is not structurally relevant, for instance `wFormatTag` in the `fmt` chunk of a WAVE file.

```
<DataModel name="Chunk">
  <String name="ckID" length="4"/>
  <Number name="cksize" size="32" >
    <Relation type="size" of="Data"/>
  </Number>
  <Blob name="Data"/>
  <Padding alignment="16"/>
</DataModel>
<DataModel name="ChunkFmt" ref="Chunk">
   <String name="ckID" value="fmt "/>
   <Block name="Data">
      <Number name="wFormatTag" size="16"/>
      <Number name="nChannels" size="16"/>
      <Number name="nSampleRate" size="32"/>
      <Number name="nAvgBytesPerSec" size="32"/>
      <Number name="nBlockAlign" size="16" />
      <Number name="nBitsPerSample" size="16"/>
   </Block>
</DataModel>
...
<DataModel name="Wav" ref="Chunk">
  <String name="ckID" value="RIFF"/>
  <String name="WAVE" value="WAVE"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="FmtChunk" ref="ChunkFmt"/>
    ...
    <Block name="DataChunk" ref="ChunkData"/>
  </Choice>
</DataModel>
```

Listing 1: WAVE Peach Pit File Format Specification

As an example, the canonical WAVE file in Figure 3 has the following virtual structure. The root chunk has start and end index $\{0, 2083\}$. The root chunk (`riff`) has three attributes, namely `ckID`, `cksize`, and `WAVEID`, and two children with indices $\{12, 35\}$ and $\{36, 2083\}$, respectively. The first child `fmt` has eight attributes namely `ckID`, `cksize`, `wFormatTag`, `nChannels`, `nSamplesPerSec`, `nAvgBytesPerSec`, `nBlockAlign`, and `wBitsPerSample`.

To construct the virtual structure, a file format specification and a parser is required. Given the specification and the file, the parser constructs the virtual structure. For example, Peach [38] has a robust parser component called *File Cracker*. Given an input file and the file format specification, called Peach Pit, our extension of the File Cracker precisely parses and decomposes the file into chunks and attributes and provides the boundary indices and type information. Listing 1 shows a snippet of the Peach Pit for the WAV file format. In this specification, we can specify the order, type, and structure of chunks and attributes in a valid WAV file. In Section 4 we explain how this specification can be constructed.

3.2. Smart Mutation Operators

Based on this virtual input structure, we define three generic structural mutation operators – *smart deletion*, *smart addition* and *smart splicing*.
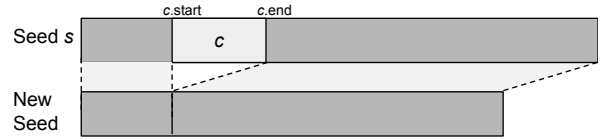
Smart Deletion. Given a seed file s, choose an arbitrary chunk c and delete it. The SGF copies the bytes following the end-index of the chosen chunk c to the start-index of c, revises the indices of all affected chunks accordingly. For instance, to delete the `fmt`-chunk in our canonical WAVE file, the stored bits in the index range $[36, 2083]$ are memcpy'd to index 12. The indices in the virtual structure of the new WAVE file are revised. For instance, the `riff`-chunk's end index is revised to 2048.
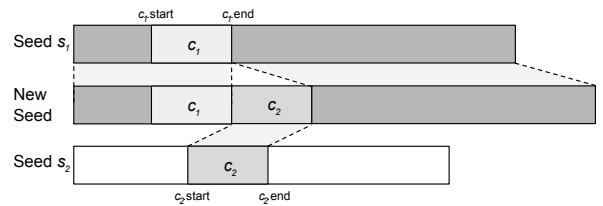
Smart Addition. Given a seed file s_1, choose an arbitrary second seed file s_2, choose an arbitrary chunk c_2 in s_2, and add it after an arbitrary existing chunk c_1 in s_1 that has a parent of the same type as c_2 (i.e., $c_1.parent.type == c_2.parent.type$). The SGF copies the bytes following the end-index of c_1 to a new index where the length of the new chunk c_2 is added to the current end-index of the c_1 in the given seed file s_1. Then, the SGF copies the bytes between start- and end-index of c_2 in the second seed file s_2 to the end-index of the existing chunk c_1 in the given seed file s_1. Finally, all affected indices are revised in the virtual structure representing the generated input.
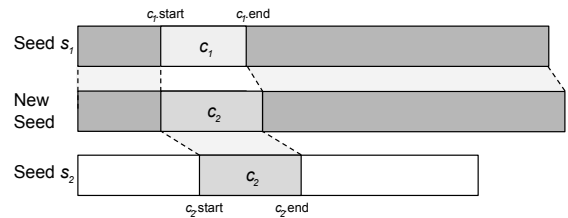
Smart Splicing. Given a seed file s_1, choose an arbitrary chunk c_1 in s_1, choose an arbitrary second seed file s_2, choose an arbitrary chunk c_2 in s_2 *such that c_1 and c_2 have the same type*, and substitute c_1 with c_2. The SGF copies the bytes following the end-index of c_1 to a new index where the length of the new chunk c_2 is added to the current end-index of the c_1 in the given seed file s_1. Then, the SGF copies the bytes between start- and end-index of c_2 in the second seed file s_2 to the end-index of the existing chunk c_1 in the given seed file s_1. Finally, all affected indices are revised in the virtual structure representing the generated input.

Maintaining validity. The files generated by applying structural mutation operators have a higher degree of validity than files generated by applying bit-level mutation operators. The specification of *immutable attributes* allows the smart greybox fuzzer to apply bit-level mutation operators only to indices of *mutable attributes* (which are not structurally relevant), increasing the likelihood to generate valid files. However, there is no guarantee that our structural mutation operators maintain the validity of a file. For instance, in our motivating example the Peach Pit format specification may allow to add or delete `fmt` chunks while strictly speaking the formal WAVE format specification allows only exactly one `fmt` chunk. Nevertheless, it was our relaxed specification which allowed finding the vulnerability in the first place (it requires two `fmt` chunks to be present). In summary, *strict* validity is not always desirable while a high degree of validity is necessary to reach beyond the parser code. This is a critical advantage of our lightweight virtual structure design.

3.3. Smart Mutation

During *smart mutation*, new inputs are generated by applying structural as well as simple mutation operators to the chosen seed file (cf. MUTATE_INPUT in Alg. 1). In the following, we discuss the challenges and opportunities of smart mutation.

3.3.1. Stacking Mutations. To generate interesting test inputs, it might be worthwhile to apply several structural (high level) and bit-level (low level) mutation operators together. In mutation-based fuzzing, this is called *stacking*. Bit-level mutation operators can easily be stacked in arbitrary order, knowing only the start- and end-index of the file. When data of length x is deleted, we subtract x from the end-index. When new data of length x is added, we add x to the new file's end-index.

However, it is not trivial to stack structural mutation operators. For each structural mutation, both the file itself and the virtual structure representing the file must be updated consistently. For instance, the deletion of a chunk will affect the end-indices of all its parent chunks, and the indices of every chunk "to the right" of the deleted chunk (i.e., chunks with a start-index that is greater than the deleted chunk's end-index). Our implementation AFLSMART makes a copy of the seed's virtual structure and stacks mutation operators by applying them consistently to both, the virtual structure and the file itself. This allows us to stack structural (high-level) mutation operators. Furthermore, if a bit-level (low-level) mutation operation cannot be translated into a mutation of the input structure, e.g., because bytes are deleted over chunk-boundaries, the mutation is not applied.

3.3.2. Deferred Parsing. In our experiments, we observed that constructing the virtual structure for a seed input incurs substantial costs. The appeal of coverage-based greybox fuzzing (CGF) and the source of its success is its *efficiency* [2]. Generating and executing an input is in the order of

a few milliseconds. However, we observed that parsing an input takes generally in the order of seconds. For instance, the construction of the virtual structure for a 218-byte PNG file takes between two and three seconds. If SGF constructs the virtual structure for every seed input that is discovered, SGF may quickly fall behind traditional greybox fuzzing despite all of its "smartness".

To overcome this scalability challenge, we developed a scheme that we call *deferred parsing*, which contributed substantially to the scalability of our tool AFLSMART. We construct the virtual structure of a seed input with a certain probability p that depends on the current time to discover a new path. Let t be the time since the last discovery of a new path. Let s be the current seed chosen by CHOOSENEXT in Line 2 of greybox fuzzing Algorithm 1 and assume that the virtual structure for s has not been constructed, yet. Given a threshold ϵ, we compute the probability $prob_{virtual}(s)$ to construct the virtual structure of s as

$$prob_{virtual}(s) = min\left(\frac{t}{\epsilon}, 1\right)$$

In other words, the probability $prob_{virtual}(s)$ to construct the virtual structure for the seed s increases as the time t since the last discovery increases. Once $t \geq \epsilon$, we have $prob_{virtual}(s) = 100\%$.

Our deferred parsing optimization is inspired by the following intuition. Without input aware greybox fuzzing as in AFLSMART, AFL may generate many invalid inputs which repeatedly traverse a few short paths in an application (typically program paths which lead to rejection of the input due to certain parse error). If more of such invalid inputs are generated, the value of t, the time since last discovery of a new path, is slated to increase. Once t increases beyond a threshold ϵ, we allow AFLSMART to construct the virtual structure. If however, normal AFL is managing to generate inputs which still traverse new paths, t will remain small, and we will not incur the overhead of creating a virtual structure. The deferred parsing optimization thus allows AFLSMART to achieve input format-awareness without sacrificing the efficiency of AFL.

3.4. Validity-based Power Schedule

A *power schedule* determines how much energy is assigned to a given seed during coverage-based greybox fuzzing [2]. The *energy* for a seed determines how much time is spent fuzzing that seed when it is chosen next (cf. ASSIGNENERGY in Alg. 1). In the literature, several power schedules have been introduced. The original power schedule of AFL [31] assigns more energy to smaller seeds with a lower execution time that have been discovered later. The gradient descent-based power schedule of AFLFAST [2] assigns more energy to seeds exercising low-frequency paths.

In the following, we define a simple validity-based power schedule. Conventionally, validity is considered as a boolean variable: Either a seed is valid, or it is not. However, we suggest to consider validity as a ratio: A file can be

valid to a certain degree. The *degree of validity* $v(s)$ of a seed s is determined by the parser that constructs the virtual structure. If all of the file can be parsed successfully, the degree of validity $v(s) = 100\%$. If only 65% of s can be parsed successfully, its validity $v(s) = 65\%$. The virtual structure for a file that is partially valid is also only partially constructed. To this partial structure, one chunk is added that spans the unparsable remainder of the file.

Given the seed s, the *validity-based power schedule* $p_v(s)$ assigns energy as follows

$$p_v(s) = \begin{cases} 2p(s) & \text{if } v(s) \geq 50\% \text{ and } p(s) \leq \frac{U}{2} \\ p(s) & \text{if } v(s) < 50\% \\ U & \text{otherwise} \end{cases} \quad (1)$$

where $p(s)$ is the energy assigned to s by the traditional greybox fuzzer's (specifically AFL's) original power schedule and U is a maximum energy that can be assigned by AFL. This power schedule implements a *hill climbing meta-heuristic* that *always* assigns twice the energy to a seed that is at least 50% valid and has an original energy $p(s)$ that is at most half the maximum energy U.

The validity-based power schedule assigns more energy to seeds with a higher degree of validity. First, the utility of the structural mutation operators increases with the degree of validity. Secondly, the hope is that more valid inputs can be generated from already valid inputs. The validity-based power schedule implements a *hill climbing meta-heuristic* where the search follows a gradient descent. A seed with a higher degree of validity will *always* be assigned higher energy than a seed with a lower degree of validity.

4. File Format Specification

The quality of file format specifications is crucial to the effectiveness and efficiency of smart greybox fuzzing. However, manually constructing such high-quality specifications of highly-structured and complicated file formats is normally criticized as a time-consuming and error-prone task. In this work, we have done an extensive research on many popular file formats (e.g., document, video, audio, image, executable and network packet files) and found the key insights based on which users can write specifications in a systematic way. These key insights explain the common structures of file formats. On the other hand, they also show the correlations between the completeness & preciseness of data models and the success of smart greybox fuzzing.

4.1. Insight-1. Chunk inheritance

Most file formats are composed of data chunks which normally share a common structure. Like an abstract class in Java and other object-oriented programming languages (e.g., C++ and C#), to write an input specification we start by modelling a generic chunk containing attributes that are shared across all chunks in the file format. Then, we model the concrete chunks which inherit the attributes from the

generic chunk. Hence, we only need to insert/modify chunk-specific attributes.

```
<DataModel name="Chunk">
 <String name="ckID" length="4" padCharacter=" "/>
 <Number name="cksize" size="32">
  <Relation type="size" of="Data"/>
 </Number>
 <Blob name="Data"/>
 <Padding alignment="16"/>
</DataModel>
```

Listing 2: Generic Chunk Model

```
<DataModel name="ChunkFmt" ref="Chunk">
    <String name="ckID" value="fmt "  token="true"
    />
    <Block name="Data">
        <Number name="wFormatTag" size="16"/>
        <Number name="nChannels" size="16"/>
        <Number name="nSampleRate" size="32"/>
        <Number name="nAvgBytesPerSec" size="32"/>
        <Number name="nBlockAlign" size="16" />
        <Number name="nBitsPerSample" size="16"/>
    </Block>
</DataModel>
```

Listing 3: Format Chunk Model

Listing 2 and Listing 3 show an example of how the chunk inheritance can be applied to the input specification of the WAVE audio file format. The generic chunk model in Listing 2 specifies that each chunk has its chunk identifier, chunk size and chunk data in which the chunk size constraints the actual length of the chunk data. Moreover, each chunk could have padded bytes at the end to make it word (2 bytes) aligned. Listing 3 shows the model of a *format* chunk, a specific data chunk in WAVE file, which inherits the chunk size and padding attributes from the generic chunk. It only models chunk-specific attributes like its string identifier and what are stored inside its data.

People normally have a big concern that they need to spend lots of time reading the standard specification of a file format (which can be hundreds of pages long) to understand this high-level hierarchical chunks structure. However, we find that there exist Hex editor tools like 010Editor [28] which can detect the file format and quickly decompose a sample input file into chunks with all attributes. The tool currently supports 114 most common file formats (e.g., PDF, MPEG4, AVI, ZIP, JPEG) [29].

Figure 5 is a screenshot of 010Editor displaying a WAVE file. The top part of the screen shows the raw data in both Hexadecimal and ASCII modes. The bottom part is the decomposed components including chunks' headers, and chunks' data.

4.2. Insight-2. Specification completeness

As explained in Section 3, smart greybox fuzzing supports structural mutation operators that work at chunk level. So we are not required to specify all attributes inside a chunk. We can start with a coarse-grained specification
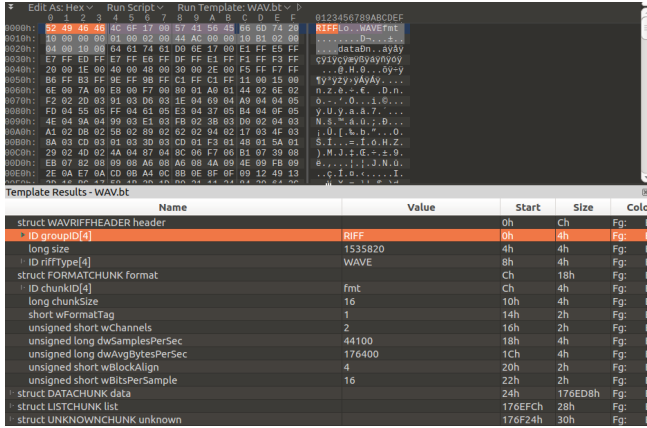
Figure 5: Analyzing file structure using 010Editor

and gradually make it more complete. Listing 4 shows a simplified definition of the format chunk in which we only specify the chunk identifier and we do not define what are the children attributtes in its data. The chunk data is considered as a "blob" which can contain anything as long as its size is consistent with the chunk size.

```
<DataModel name="ChunkFmt" ref="Chunk">
   <String name="ckID" value="fmt "  token="true"
    />
</DataModel>
```

Listing 4: Simplified Format Chunk Model

Based on the this key insight and the Insight-1, one can quickly write a short yet precise file format specification. As shown in Section 5, the specification for the WAVE file format can be written in 82 lines while the specification for the PCAP network traffic file format can be written in just 24 lines. These two specifications helped smart greybox fuzzing discover many vulnerabilities which could not be found by other baseline techniques.

4.3. Insight-3. Relaxed constraints

There could be many constraints in a chunk (e.g., the chunk identifier must be a constant string, the chunk size attribute must match with the actual size or chunks must be in order). However, since the main goal of fuzzing or stress testing in general is to explore corner cases, we should relax some constraints as long as these relaxed constraints do not prevent the parser from decomposing the file. Listing 5 shows the definition of a WAVE file format. As we use the *Choice* element[1] to specify the list of potential chunks (including both mandatory and optional ones), many constraints have been relaxed. Firstly, the chunks can appear in any order. Secondly, some chunk (including mandatory chunk) can be absent. Thirdly, some unknown chunk can

1. In a Peach pit, Choice elements are used to indicate any of the sub-elements are valid but only one should be selected at a time. Reference: http://community.peachfuzzer.com/v3/Choice.html

appear. Lastly, some chunk can appear more than once. In fact, becaused this relaxed model, vulnerabilities like the one in our motivating example in our paper (Section 2) can be exposed.

```
<DataModel name="Wav">
  <String name="ckID" value="RIFF" token="true"/>
  <Number name="cksize" size="32" />
  <String name="WAVE" value="WAVE" token="true"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="FmtChunk" ref="ChunkFmt"/>
    <Block name="DataChunk" ref="ChunkData"/>
    <Block name="FactChunk" ref="ChunkFact"/>
    <Block name="SintChunk" ref="ChunkSint"/>
    <Block name="WavlChunk" ref="ChunkWavl"/>
    <Block name="CueChunk" ref="ChunkCue"/>
    <Block name="PlstChunk" ref="ChunkPlst"/>
    <Block name="LtxtChunk" ref="ChunkLtxt"/>
    <Block name="SmplChunk" ref="ChunkSmpl"/>
    <Block name="InstChunk" ref="ChunkInst"/>
    <Block name="OtherChunk" ref="Chunk"/>
  </Choice>
</DataModel>
```

Listing 5: WAVE File Format Specification

4.4. Insight-4. Reusability

Unlike specifications of program behaviours which are program specific and hardly reusable, a file format specification can be used to fuzz all programs taking the same file format. We believe the benefit of finding new vulnerabilities far outweighs the cost of writing input specifications. In Section 5 and Section 6, we show that our smart greybox fuzzing tool have used specifications of 10 popular file formats (PDF, AVI, MP3, WAV, JPEG, JPEG2000, PNG, GIF, PCAP, ELF) to discover more than 40 vulnerabilities in heavily-fuzzed real-world software packages. Notably, based on the key insights we have presented, it took one of us only five (5) working days to complete these 10 specifications.

5. Experimental Setup

To evaluate the effectiveness and efficiency of smart greybox fuzzing, we conducted several experiments. We implemented our technique by extending the existing greybox fuzzer AFL and call our smart greybox fuzzer AFLSMART. To investigate whether input-structure-awareness indeed improves the vulnerability finding capability of a greybox fuzzer, we compare AFLSMART with two traditional greybox fuzzers AFL [31] and AFLFAST [2]. To investigate whether smart blackbox fuzzer (given the same input model) could achieve a similar vulnerability finding capability, we compare AFLSMART with the smart blackbox fuzzer Peach [38]. We also compare AFLSMART with VUZZER [25]. The objective of VUZZER is similar to AFLSMART, it seeks to tackle the challenges of structured file formats for greybox fuzzing, yet without input specifications, using taint analysis and control flow analysis.

5.1. Research Questions

RQ-1. *Is smart greybox fuzzing more effective and efficient than traditional greybox fuzzing?* Specifically, we investigate whether AFLSMART exposes more unique crashes than AFL/AFLFAST in 24 hours, and in the absence of crashes whether AFLSMART explores more paths than AFL/AFLFAST in the given time budget.

RQ-2. *Is smart greybox fuzzing more effective and efficient than smart blackbox fuzzing?* Specifically, we investigate whether AFLSMART exposes more unique crashes than Peach in 24 hours, and in the absence of crashes whether AFLSMART explores more paths than Peach in the given time budget.

RQ-3. *Is smart greybox fuzzing more effective than taint analysis-based greybox fuzzing?* Specifically, we investigate the number of bugs found by each technique individually and all together.

5.2. Implementation: AFLSMART

AFLSMART extends AFL by adding and modifying four components, the File Cracker, the Structure Collector, the Energy Calculator and the Fuzzer itself. The overall architecture is shown in Figure 6. While currently integrated with Peach, we designed AFLSMART such that it provides a general framework that allows integrating other input parsers and to define further structural mutation operators.
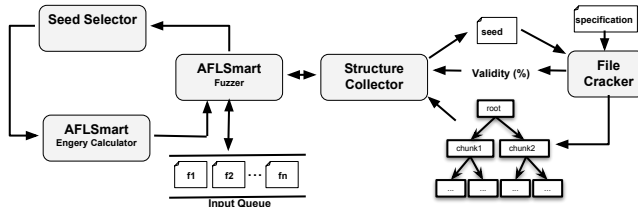

Figure 6: Architecture of AFLSMART

AFLSMART File Cracker parses an input file and decomposes it into data chunks and data attributes. It also calculates the validity of the input file based on how much of the file can be parsed. In this prototype, we implement the File Cracker by modifying the Cracker component of the smart blackbox fuzzer Peach (Community version) [38] which fully supports highly-structured file formats such as PNG, JPEG, GIF, MP3, WAV and AVI.

AFLSMART Structure Collector connects the core AFLSMART Fuzzer and the File Cracker component. When the Fuzzer requests structure information of the current input to support its operations (e.g., smart mutations), it passes the input to the Structure Collector for collecting the validity and the decomposed chunks and attributes. This component provides a generic interface to support all File Crackers – our current Peach-based File Cracker and new ones. It is also worth noting that AFLSMART Fuzzer only collects these information once and saves them for future uses.

AFLSMART Energy Calculator implements the validity-based power schedule as discussed in Section 3. Hence, AFLSMART assigns more energy to inputs which are more syntactically valid. Specifically, we apply a new formula to the *calculate_score* function of AFLSMART.

AFLSMART Fuzzer contains the most critical changes to make AFLSMART effective. In this component, we design and implement the virtual structure which can represent input formats in a hierarchical structure. Based on this core data structure, all AFLSMART mutation operations which work at chunk levels are implemented. We also modify the *fuzz_one* function of AFL to support our important optimizations – deferred parsing and stacking mutations (Section 3).

Note that our changes do not impact the instrumentation component of AFL. As a result, we can use AFLSMART to fuzz program binaries provided the binary is instrumented using a tool like DynamoRio [4] and the instrumented code can be processed by AFL. Such a binary fuzzing approach has been achieved in the WinAFL tool[2] for Windows binaries. AFLSMART works well with such binary fuzzing tools.

5.3. Subject Programs

We did a rigorous search for suitable benchmarks to test AFLSMART and the chosen baselines. We evaluated the techniques using both large real-world software packages and a benchmark previously used in VUZZER paper. We did not use the popular LAVA benchmarks [10] because the LAVA-M subjects (*uniq, base64, md5sum, who*) do not process structured files while the small *file* utility in LAVA-1 takes any file, regardless of its file format, and determines the file type.

In the comparison with AFL, AFLFAST and Peach (RQ-1 and RQ-2), we selected the newest versions (at the time of our experiments) of 11 experimental subjects from well-known open source programs which take ten (6) highly-structured file formats – executable binary file (ELF), image files (PNG, JPEG, JP2 (JPEG2000)), audio/video files (WAV, AVI). All of them have been well tested for many years. Notably, five (5) media processing libraries (FFmpeg[3], LibPNG[4], LibJpeg-Turbo[5], ImageMagick[6], and OpenJPEG[7]) have joined the Google OSS-Fuzz project[8] and they are continuously tested using the state-of-the-art fuzzers including AFL and LibFuzzer. LibAV[9], WavPack [10] and Libjasper[11] are widely-used libraries and tools for image, audio and video files processing and streaming. Binutils[12]

---

2. https://github.com/ivanfratric/winafl
3. https://github.com/FFmpeg/FFmpeg
4. https://github.com/glennrp/libpng
5. https://github.com/libjpeg-turbo
6. https://github.com/ImageMagick/ImageMagick
7. https://github.com/uclouvain/openjpeg
8. https://github.com/google/oss-fuzz
9. https://github.com/libav/libav
10. https://github.com/dbry/WavPack
11. https://github.com/mdadams/jasper
12. https://www.gnu.org/software/binutils/

TABLE 1: Subject Programs and File Formats. VUZZER subjects are at the bottom.

Program	Description	Size (LOC)	Test driver	Format	Option
Binutils	Binary analysis utilities	3700 K	`readelf`	**ELF**	`-agteSdcWw --dyn-syms -D @@`
Binutils	Binary analysis utilities	3700 K	`nm-new`	**ELF**	`-a -C -l --synthetic @@`
LibPNG	Image processing	111 K	`pngimage`	**PNG**	`@@`
ImageMagick	Image processing	385 K	`magick`	**PNG**	`@@ /dev/null`
LibJPEG-turbo	Image processing	87 K	`djpeg`	**JPEG**	`@@`
LibJasper	Image processing	33 K	`imginfo`	**JPEG**	`-f @@`
FFmpeg	Video/Audio/Image processing	1100 K	`ffmpeg`	**AVI**	`-y -i @@ -c:v mpeg4 -c:a out.mp4`
LibAV	Video/Audio/Image processing	670 K	`avconv`	**AVI**	`-y -i @@ -f null -`
LibAV	Video/Audio/Image processing	670 K	`avconv`	**WAV**	`-y -i @@ -f null -`
WavPack	Lossless Wave file compressor	47 K	`wavpack`	**WAV**	`-y @@ -o out_dir`
OpenJPEG	Image processing	115 K	`decompress`	**JP2**	`-y @@ -o out_dir`
LibJasper	Image processing	33 K	`jasper`	**JP2**	`-y @@ -o out_dir`
mpg321	Command line MP3 player	5 K	`mpg321`	**MP3**	`--stdout @@`
gif2png+libpng	Image converter	36 K	`gif2png`	**GIF**	`@@`
pdf2svg+libpoppler	PDF to SVG converter	92 K	`pdf2svg`	**PDF**	`@@ out.svg`
tcpdump+libpcap	Network traffic analysis	102 K	`tcpdump`	**PCAP**	`-nr @@`
tcptrace+libpcap	TCP connection analysis	55 K	`tcptrace`	**PCAP**	`@@`
djpeg+libjpeg	Image processing	37 K	`djpeg`	**JPEG**	`@@`

is a set of utilities for analyzing binary executable files. It is installed on almost all Linux-based machines.

To compare with VUZZER (RQ-3), we chose the same benchmark used in the paper. The benchmark includes old versions of six (6) popular programs on Ubuntu 14.04 32-bit: mpg321 (v0.3.2), gif2png (v2.5.8), pdf2svg (v0.2.2), tcp-dump (v4.5.1), tcptrace (v6.6.7), and djpeg (v1.3.0). These subjects take MP3, GIF, PDF, PCAP and JPEG files as inputs. It is worth noting that VUZZER has not supported 64-bit environment yet.

Table 1 shows the full list of programs and their information. Note that the sizes of subject programs are calculated by `sloccount`.[13] Moreover, to increase the reproducibility of our experiments, we also provide the exact command options we used to run the subject programs. In the experiments to answer RQ-1 and RQ-2, we tested two programs for each file format to mitigate subject bias.

5.4. Corpora, Dictionaries, and Specifications

Format specification. AFLSMART leverages file format specifications to construct the virtual structure of a file. These specifications are developed as Peach Pits.[14] In our experiment, we used ten file format specifications (see Table 2). While the specification of the WAV format is a modification of a free Peach sample[15], we developed other Peach pits from scratch. AFLSMART and Peach are provided with the same file format specifications (i.e., Peach pits).

Seed corpus. In order to construct the initial seed files, we leveraged several sources. For PNG and JPEG images, we used the image files that are available as test files in their respective code repositories. For ELF files, we collected program binaries from the *bin* and */user/bin* folders on the host machine. For other file formats, we downloaded seed

13. https://www.dwheeler.com/sloccount/
14. http://community.peachfuzzer.com/v3/PeachPit.html
15. http://community.peachfuzzer.com/v3/TutorialFileFuzzing/

TABLE 2: File Format Specifications and Seed Corpora

Format	File Format Specification Length (#Lines)	Time spent	Seed Corpus #Files	Avg. size
ELF	90 lines	4 hours	21	100 KB
PNG	128 lines	4 hours	51	4 KB
JPEG	92 lines	4 hours	8	5.5 KB
WAV	82 lines	1 hour	11	500 KB
AVI	124 lines	4 hours	10	430 KB
JP2	144 lines	4 hours	10	35 KB
PDF	84 lines	4 hours	10	140 KB
GIF	108 lines	4 hours	10	12 KB
PCAP	24 lines	4 hours	5	11 KB
MP3	90 lines	4 hours	10	201 KB

inputs from websites keeping sample files (WAV[16], AVI[17], JP2[18], PCAP[19], MP3[20], GIF[21] and PDF[22]). Table 2 shows the size of the input corpus we used for each file format. All fuzzers are provided with the same initial seed corpus.

Dictionary. We developed dictionaries for four (4) file formats (ELF, WAV, AVI, and JP2); AFL (and AFLSMART) already provides dictionaries for PNG and JPEG image formats. The dictionaries were written by simply crafting the tokens (e.g., signatures, chunk types) from the same specifications/documents based on which we developed the Peach Pit file format specifications. Both AFLSMART and AFL were run with dictionaries.

Reproducibility. To ensure the reproducibility of our experiments, we will make AFLSMART open source and provide the seed corpora, dictionaries, and Peach Pits used.

16. https://freewavesamples.com/source/roland-jv-2080
17. http://www.engr.colostate.edu/me/facil/dynamics/avis.htm
18. http://samples.ffmpeg.org/
19. https://wiki.wireshark.org/SampleCaptures
20. https://www.magnac.com/sounds.shtml
21. https://people.sc.fsu.edu/ jburkardt/data/gif/gif.html
22. https://www.pdfa.org/isartor-test-suite/

5.5. Infrastructure

Computational Resources. We have different setups for two sets of experiments. In the first set of experiments to compare AFLSMART with AFL, AFLFAST, and Peach we used machines with an Intel Xeon CPU E5-2660v3 processor that has 56 logical cores running at 2.4GhZ. Each machine runs Ubuntu 16.04 (64 bit) and has access to 64GB of main memory. All fuzzers have the same time budget (24 hours), the same computational resources, and are started with the same seed corpus with the same dictionaries. Peach and AFLSMART also use the same Peach Pits.

In the comparison with VUZZER, as VUZZER has not supported 64-bit environment yet, we set up a virtual machine (VM) having the same settings reported in the paper – a Ubuntu 14.04 LTS system equipped with a 32-bit 2-core Intel CPU and 4 GB RAM. Both VUZZER and AFLSMART are started with the same seed corpus.

Experiment repetition. To mitigate the impact of randomness, for each subject program we run five (5) isolated instances of each of AFL, AFLFAST, AFLSMART, and Peach in parallel. We emphasize that none of the instances share the same queue. Specifically, Peach does not support the shared queue architecture (i.e., parallel fuzzing mode in AFL[23]).

Measurement in AFL-based fuzzers. The greybox fuzzers AFL, AFLFAST, and AFLSMART already provide the number of explored paths in five-second intervals in a file called plot_data. This allows us to plot these quantities over time. To compute the number of unique bugs found, we used a call stack-based bucketing approach [9] to analyze and group the discovered bugs. Crashes that have the exact the same call stack are in the same group. We selected one representative from each group for bug reporting purposes.

Measurement in Peach. Peach does not keep the generated test cases. It only stores bug-triggering inputs which complicates our measurement of the number of paths explored. Hence, we modified Peach such that we could collect all test cases which Peach generates during a 24-hour run. Then, we use the *afl-cmin*[24] – a corpus minimization utility in the AFL toolset to find the smallest subset of files in the generated test cases that still trigger the full range of instrumentation data points. To achieve a fair comparison, we also use the same *afl-cmin* to minimize the test cases generated by AFL, AFLFAST and AFLSMART. These results are reported in the fourth column (#Min-set) of the Table 3

6. Experimental Results

RQ.1 SGF Versus Traditional Greybox Fuzzing

In terms of discovered number of paths, AFLSMART *clearly outperforms both* AFL *and* AFLFAST. AFLSMART discovered more paths in ten (10) out of twelve (12) subjects. In the two larger subjects, ffmpeg and avconv (taking

23. https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt
24. https://github.com/mirrorer/afl/blob/master/afl-cmin

TABLE 3: Average number of paths discovered, the minimal sets of test cases calculated by afl-cmin, crashes found, and unique bugs discovered in 5 runs after 24 hours.

Binary	Fuzzer	#Paths	#Min-set	#Crashes	#Bugs
readelf ELF	AFL	14855	6285	15	3
	AFLFAST	16048	6422	22	3
	Peach	N/A	1202	0	0
	AFLSMART	16236	7002	19	3
nm-new ELF	AFL	10201	4283	33	1
	AFLFAST	10159	3995	45	1
	Peach	N/A	454	0	0
	AFLSMART	8981	3885	34	2
pngimage PNG	AFL	5280	2324	0	0
	AFLFAST	5663	2294	0	0
	Peach	N/A	395	0	0
	AFLSMART	6497	2560	1	1
magick PNG	AFL	6434	2696	0	0
	AFLFAST	6249	2668	0	0
	Peach	N/A	66	0	0
	AFLSMART	6860	2861	0	0
djpeg JPEG	AFL	3661	1275	0	0
	AFLFAST	3778	1264	0	0
	Peach	N/A	342	0	0
	AFLSMART	4005	1351	0	0
imginfo JPEG	AFL	1681	967	18	2
	AFLFAST	1437	759	44	2
	Peach	N/A	53	0	0
	AFLSMART	1812	1003	58	2
ffmpeg AVI	AFL	2783	1340	0	0
	AFLFAST	3378	1547	0	0
	Peach	N/A	1413	0	0
	AFLSMART	8485	3582	2	1
avconv AVI	AFL	4980	1205	213	3
	AFLFAST	4900	1209	218	3
	Peach	N/A	849	0	0
	AFLSMART	13549	3328	503	3
avconv WAV	AFL	14849	4271	0	0
	AFLFAST	14617	4209	0	0
	Peach	N/A	867	0	0
	AFLSMART	20616	6418	13	3
wavpack WAV	AFL	1724	425	59	1
	AFLFAST	1950	460	48	1
	Peach	N/A	339	0	0
	AFLSMART	1998	537	191	5
decompress JPEG2000	AFL	6615	1984	0	0
	AFLFAST	6767	2030	0	0
	Peach	N/A	389	0	0
	AFLSMART	6503	1950	16	3
jasper JPEG2000	AFL	2624	1049	220	6
	AFLFAST	2298	954	156	5
	Peach	N/A	215	0	0
	AFLSMART	3957	1582	944	10

AVI files), AFLSMART explored 200% more paths than AFL and AFLFAST. The same improvement can be observed in the minimized sets of test cases (#Min-set) as well. AFLSMART performed a bit worse than AFL and AFLFAST (in terms of path exploration) in a ELF-parsing subject in Binutils (nm-new) and an OpenJPEG utility (decompress). For these two subjects, AFLSMART achieved similar path coverage in the first six (6) hours after which AFL and AFLFAST started outperforming AFLSMART (see Figure 7).

In terms of bug finding, AFLSMART *discovered bugs in 10 subjects while* AFL *and* AFLFAST *could not detect bug in four of them (ffmpeg, pngimage, decompress and avconv (taking WAV files)).* After analyzing the crashes, we

TABLE 4: Bug reports. Assertion Failure (AF), Aborted (AB), Divide-by-Zero (DZ), Heap/Stack Overflow (OF), Null Pointer Reference (NP)

Subject	Bug-ID	Type	AFL	AFLFAST	Peach	AFLSMART
WavPack	CVE-2018-10536	OF	✗	✗	✗	✓
	CVE-2018-10537	OF	✗	✗	✗	✓
	CVE-2018-10538	OF	✗	✗	✗	✓
	CVE-2018-10539	OF	✗	✗	✗	✓
	CVE-2018-10540	OF	✓	✓	✗	✓
Binutils	Bugzilla-23062	AF	✓	✓	✗	✓
	Bugzilla-23063	AF	✓	✓	✗	✓
	CVE-2018-10372	OF	✓	✓	✗	✓
	CVE-2018-10373	NP	✓	✓	✗	✓
	Bugzilla-23177	OF	✗	✗	✗	✓
LibPNG	CVE-2018-13785	DZ	✗	✗	✗	✓
Libjasper	Issue-174	AF	✓	✓	✗	✓
	Issue-175	AF	✓	✓	✗	✓
	Issue-182-1	OF	✗	✗	✗	✓
	Issue-182-2	NP	✗	✗	✗	✓
	Issue-182-3	OF	✗	✗	✗	✓
	Issue-182-4	NP	✗	✗	✗	✓
	Issue-182-5	OF	✓	✓	✗	✓
	Issue-182-6	AF	✓	✓	✗	✓
	Issue-182-7	AF	✓	✓	✗	✓
	Issue-182-8	AB	✓	✓	✗	✓
	Issue-182-9	AF	✓	✓	✗	✓
	Issue-182-10	AF	✓	✗	✗	✓
OpenJPEG	Email-Report-1	OF	✗	✗	✗	✓
	Email-Report-2	OF	✗	✗	✗	✓
	Issue-1125	AF	✗	✗	✗	✓
LibAV	Bugzilla-1121	OF	✗	✗	✗	✓
	Bugzilla-1122	OF	✗	✗	✗	✓
	Bugzilla-1123	OF	✗	✗	✗	✓
	Bugzilla-1124	OF	✓	✓	✗	✓
	Bugzilla-1125	DZ	✓	✓	✗	✓
	Bugzilla-1127	OF	✓	✓	✗	✓
FFmpeg	Email-Report-3	DZ	✗	✗	✗	✓
TOTAL			16	15	0	33

reported 33 zero-day bugs found by AFLSMART out of which only 16 bugs were found by AFL and AFLFAST. Vice versa, all zero-day bugs that AFL and AFLFAST found were also found by AFLSMART. Hence, AFLSMART discovered twice as many bugs as AFL/AFLFAST. Table 4 shows the detailed bugs found by AFLSMART and the baseline. 17 bugs are heap & stack buffer overflows (many of them are buffer overwrites) which are known to be easily exploitable. The maintainers of these programs have fixed 12 bugs we reported. The MITRE corporation[25] has assigned eight (8) CVEs to the most critical vulnerabilities.

The main reason why AFL and AFLFAST could not find many bugs, meanwhile AFLSMART did, in subjects like FFmpeg, LibAV, WavPack, and OpenJPEG is that these programs take in highly structured media files (e.g., image, audio, video) in which the data chunks must be placed in order at correct locations. This is very challenging for traditional greybox fuzzing tools like AFL and AFLSMART. In addition to the motivating example (CVE-2018-10536 and CVE-2018-10537), we analyze in depth few more critical vulnerabilities found by AFLSMART to explain the challenges.

CVE-2018-10538: Heap Buffer Overwrite. The *buffer overwrite* is caused by two integer overflows and insufficient memory allocation. To construct an exploit, we need to craft a *valid WAVE file* that contains the mandatory `riff`, `fmt`, and `data` chunks. Between the `fmt` and `data` chunk, we add an *additional unknown chunk* (i.e., that is neither fmt, data, ..) with `cksize ≥ 0x80000000`.

During parsing the file, WavPack enters the "unknown chunk" handling code shown in Figure 8. It reads the specified chunk size from the `chunk_header` struct and stores it as a 32-bit *signed* integer. Since $ckSize \geq 2^{31}$, the assignment in `riff.c:288` overflows, such that `bytes_to_copy` contains a negative value. The memory allocation function `malloc` takes only unsigned values causing a second overflow to a smaller positive number. When `DoReadFile` attempts to read more information from the WAVE file, there is not enough memory being allocated, resulting in a memory overwrite that can be controlled by the attacker. This vulnerability (CVE-2018-10538) was patched by aborting when `bytes_to_copy` is negative.

OpenJPEG-1: Heap Buffer Overread & Overwrite.

The buffer overread (lines 617-619) and overwrite (lines 629-631) (see Figure 9) are caused by a missing check of the actual size (width and height) of the three color streams (red, green, and blue). Without this check, the code assumes that all the three streams have the same size and it uses the same bound value (*max*) to access the buffers. To construct an exploit, we need to craft a *valid* JP2 (JPEG2000) file that contains three color streams having different sizes by "swapping" the whole stream(s) from one valid JP2 file and place it/them in the correct position(s) in another valid JP2 file. Without the structural information, traditional greybox fuzzing is unlikely to do such a precise swapping.

RQ.2   SGF Versus Smart Blackbox Fuzzing

Given the same input format specifications, AFLSMART *clearly outperforms Peach in all twelve (12) subjects* (see Table 3 and Table 4). AFLSMART generated up to an order of magnitude meaningful test cases (see #Min-set column in Table 3) and discovered 33 zero-day bugs while Peach could not find a single vulnerability .[26]

Apart from the difficulty to discover zero-day bugs in the heavily-fuzzed benchmarks, we explain these results by the lack of coverage feedback mechanism in Peach. The smart blackbox fuzzer treats all test cases at all stages equally. There is no evolution of a seed corpus. Instead, there is a simple enumeration of files that are valid w.r.t. the provided specification. This is a well-kown limitation of Peach. Recently Lian et. al [19] have tried to tackle this problem by applying LLVM passes and designing a feedback mechanism for Peach. The tool is not available for further comparison and analysis.

26. Unlike for the AFL-based fuzzers, Peach does not produce data that allows us to plot the number of paths discovered over time in Figure 7.
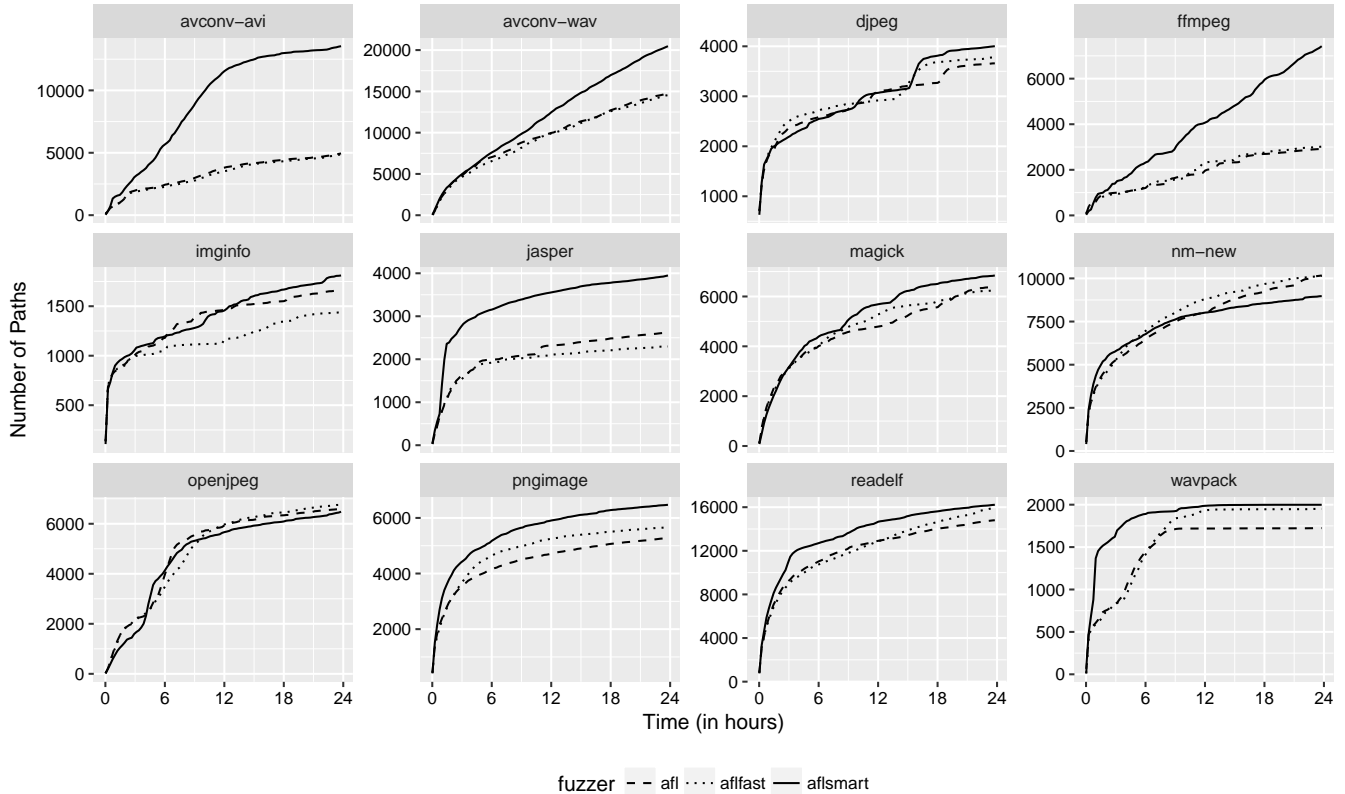
Figure 7: Number of paths discovered over time for AFL, AFLFAST, and AFLSMART

```
286  else {                        // just copy unknown chunks to output file
287
288  int bytes_to_copy=(chunk_header.ckSize+1) &  1L;
289  char *buff=malloc(bytes_to_copy);
...
296  if (!DoReadFile(infile,buff,bytes_to_copy,..)) {
```

Figure 8: Showing `cli/riff.c` @ revision `0a72951`

A second explanation is the completeness of the file format specification. The performance of Peach substantially depends on the precision and completeness of the file format specification. Peach might need more detailed input models in which (almost) all chunks and attributes are specified with exact data types to generate more interesting files. In contrast, AFLSMART does not require very detailed file format specifications to derive the virtual structure of a file and apply our structural mutation operators.

RQ.3 Versus Taint analysis-based Greybox Fuzzing

AFLSMART *outperforms* VUZZER *on a* VUZZER*'s benchmark*. AFLSMART found 15 bugs in all subject programs in the benchmark in which seven (7) bugs could not be found by VUZZER in *tcpdump*, *tcptrace* and *gif2png* (see Table 5. It is worth noting that all these bugs are not zero-day ones because the VUZZER benchmark contains old versions of software packages on the out-dated Ubuntu 14.04 32-bit;

```
612  r = image->comps[0].data;
613  g = image->comps[1].data;
614  b = image->comps[2].data;
...
616  for (i = 0U; i < max; ++i) {
617    *in++ = (unsigned char) * r++;
618    *in++ = (unsigned char) * g++;
619    *in++ = (unsigned char) * b++;
620  }
...
622  cmsDoTransform(transform, inbuf, outbuf, ...);
...
624  r = image->comps[0].data;
625  g = image->comps[1].data;
626  b = image->comps[2].data;
...
628  for (i = 0U; i < max; ++i) {
629    *r++ = (unsigned char) * out++;
630    *g++ = (unsigned char) * out++;
631    *b++ = (unsigned char) * out++;
632  }
```

Figure 9: Showing `common/color.c` @ rev `d2205ba`

all the bugs have been fixed. We explain these results by the limited information VUZZER can infer using taint analysis – it cannot infer the high-level structural representation of the input so it cannot do mutations at the chunk level.

We also investigate the intersection of the results. As shown in Figure 10, VUZZER and AFLSMART discov-

TABLE 5: VUZZER vs AFLSMART on VUZZER's benchmark

Application	Vuzzer		AFLSMART	
	#Crashes	#Bugs	#Crashes	#Bugs
mpg321	337	2	193	2
gif2png+libpng	127	1	54	2
pdf2svg+libpoppler	13	3	20	2
tcpdump+libpcap	3	1	149	6
tcptrace+libpcap	403	1	240	2
djpeg+libjpeg	1	1	1	1
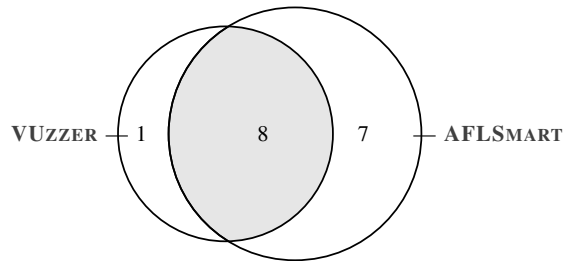
Figure 10: Venn Diagram showing the number of bugs that VUZZER and AFLSMART discover individually and together.

ered 16 bugs all together. Even though the intersection is large (AFLSMART discovered almost all bugs found by VUZZER), we believe AFLSMART and VUZZER are two potentially supplementary approaches. While AFLSMART can leverage the input structure information to systematically do mutations at the chunk level and explore new search space (which is unlikely to be done by bit-level mutations), VUZZER can leverage its taint analysis to infer features of attributes inside the newly generated inputs and mutate them effectively.

7. Case Study. Bug Hunting using AFLSMART

We conducted an extra experiment to evaluate the effectiveness of AFLSMART in a bug hunting campaign for a large and popular software package. We chose FFmpeg as our target program because this is an extremely popular and heavily-fuzzed library. Every day when we use our computers/smartphones in working time or in our leisure time, we would use at least one software powered by the FFmpeg library like a web browser (e.g., Google Chrome), a sharing video page (e.g., YouTube), or a media player (e.g., VLC). FFmpeg is heavily fuzzed; as a part of OSS-Fuzz project, it has been continuously fuzzed for years. Due to its popularity, any serious vulnerability in FFmpeg could compromise millions of systems and expose critical security risk(s).

We run five (5) instances of AFLSMART in parallel mode[27] in one week using the AVI input specification to test its functionality of converting an AVI file to a MPEG4 file (see Table 1 for the exact command). In this fuzzing campaign, AFLSMART discovered nine (9) zero-day crashing

27. https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

TABLE 6: CVEs of bugs found in FFmpeg

Subject	Bug-ID	Description	Severity
FFmpeg	CVE-2018-13301	Null pointer dereference	MEDIUM
	CVE-2018-13305	Heap buffer overwrite	HIGH
	CVE-2018-13300	Heap buffer overread	HIGH
	CVE-2018-13303	Null pointer dereference	MEDIUM
	CVE-2018-13302	Heap buffer overwrite	HIGH
	CVE-2018-12459	Assertion failure	MEDIUM
	CVE-2018-12458	Assertion failure	MEDIUM
	CVE-2018-13304	Assertion failure	MEDIUM
	CVE-2018-12460	Null pointer dereference	MEDIUM

bugs including buffer overflows, null pointer dereferences and assertion failures. All the bugs have been fixed and nine (9) CVE IDs have been assigned to them. Table 6 shows the CVEs and their severity levels based on the Common Vulnerability Scoring System version 3.0 [32]; all these nine vulnerabilities are rated from medium to high severity.

The results confirm the practical impact of smart grey-box fuzzing in testing programs taking highly-structured input files like FFmpeg. It shows that the benefit of finding new vulnerabilities outweighs the one-time effort of writing input specifications.

8. Related Work

Smart blackbox fuzzing. The stream of works that is most closely related to ours is that of smart blackbox fuzzers which leverage file format specifications to generate inputs for a program that is otherwise treated as a blackbox. In the area of smart blackbox fuzzing, input grammars have been used to generate test inputs [24]. There exist a variety of tools employing this technique, such as Peach fuzzer [38], Spike [39], Domato [33], and LangFuzz [15]. LangFuzz is a smart blackbox fuzzer that has been used to detect crashes in JavaScript engines; it uses a file format specification to mutate a given seed input and replaces code fragments with those learned from a set of parsed sample inputs. Our work on AFLSMART can be seen as integrating the format-awareness capability into coverage-based grey-box fuzzing.

Smart whitebox fuzzing. Another related stream of works is that of smart whitebox fuzzing which leverages both program structure and input structure to explore the program most effectively. Whitebox fuzzers are often based on symbolic execution engines such as KLEE [5], or S^2E [8]. Grammar-based whitebox fuzzers [12] can generate files that are valid w.r.t. a context-free grammar. Model-based whitebox fuzzing [23] enforces semantic constraints over the input structure that cannot be expressed in a context-free grammar, such as length-of relationships. In contrast to our approach, smart whitebox fuzzers require heavy machinery of symbolic execution and constraint solving.

Coverage-based greybox fuzzing. Our work builds on coverage-based greybox fuzzing (CGF) [31], [37], which is a popular and effective approach for software vulnerability detection. The AFL fuzzer [31] and its extensions [1], [2], [7], [11], [17], [18], [21], [27] constitute the most widely used embodiment of CGF. CGF is a promising middle ground between blackbox and whitebox fuzzing. Compared

to blackbox approaches, CGF uses light-weight instrumentation to guide the fuzzer to new regions of the code, and compared to whitebox approaches, CGF does not suffer from high overheads of constraint solving [3]. To the best of our knowledge, ours is the first work to propose and build an input format-aware greybox fuzzer.

Boosted greybox fuzzing. AFLFAST [2] uses Markov chain modeling to target regions that are still not generally covered by AFL. The approach discovers known bugs faster compared to standard AFL, as well as finding new bugs. AFLGO [1] performs reachability analysis to a given location or target by prioritizing seeds which are estimated to have a lower distance to the target. Angora [7] is an extension of AFL to improve its coverage that performs search based on gradient descent to solve path condition without symbolic execution. SlowFuzz [22] prioritizes inputs with a higher resource usage count for further mutation, with the objective of discovering vulnerabilities to complexity attacks. These works improve the effectiveness of greybox fuzzing along other dimensions (not input format awareness), and are largely orthogonal to our approach

Restricted mutations. Other works in the CGF area employ specific optimizations to restrict the mutations. VUzzer [25] uses data- and control-flow analysis of the test subject to detect the locations and the type of the input data to mutate or to keep constant. Steelix [18] focuses on developing customized mutation operations of *magic bytes*, e.g., the special words `RIFF`, `fmt`, or `data` in a WAVE file (see 2). SymFuzz [6] learns the dependencies in the bits in the seed input using symbolic execution in order to compute an optimal *mutation ratio* given a program under test and the seed input; the mutation ratio is the number of seed bits that are flipped in mutation-based fuzzing. These works encompass specific optimizations to restrict mutations. They do *not* inject input format awareness for generating valid inputs as is achieved by our file format aware mutation operators, or validity-based power schedules.

Greybox fuzzing and symbolic execution. T-Fuzz [21] removes sanity checks in the code that blocks the fuzzers (AFL or honggfuzz [36]) from progressing further. This, however, introduces false positives, which are then detected using symbolic execution. Driller [27] is a combination of fuzzing and symbolic execution to allow for deep exploration of program paths. In our work, we avoid any symbolic execution, and enhance the effectiveness of grey-box fuzzing without sacrificing the efficiency of AFL.

Format specification inference. Several works study file format inferencing. Lin and Zhang [20] present an approach to derive the file's input tree from the dynamic execution trace. Learn&Fuzz [14] uses neural-network-based statistical machine learning to generate files satisfying a complex format. The approach is used to fuzz Microsoft Edge browser PDF handler, and found a bug not previously found by previous approaches such as SAGE [13]. AUTOGRAM [16] uses dynamic taint analysis to derive input grammars. Such works on input format inference can potentially help input-aware fuzzers such as AFLSMART.

9. Discussion

Greybox fuzzing has been the technology of choice for practical, automated detection of software vulnerabilities. The current embodiment of greybox fuzzing in the form of the AFL fuzzer is agnostic to the input format specification. This leads to lot of time in a fuzzing campaign being wasted in generation of syntactically invalid inputs. In this work, we have brought in the input format awareness of commercial blackbox fuzzers into the domain of greybox fuzzing. This is achieved via file format aware mutations, validity-based power schedules, and several optimizations (most notably the deferred parsing optimization) which allows our AFLSMART tool to retain the efficiency of AFL. Detailed evaluation of our tool AFLSMART with respect to AFL on applications processing popular file formats (such as AVI, MP3, WAV) demonstrate that AFLSMART achieves substantially (up to 200%) higher path coverage and finds more bugs as compared to AFL. The manual effort of specifying an input format is a one-time effort, and was limited to 4 hours for each of the input formats we examined.

In future, we can extend the input file-format fuzzing of AFLSMART to input protocol fuzzing by taking into account input protocol specifications, along the lines of the state model already supported by the Peach fuzzer. This will allow us to extend AFLSMART for fuzzing of reactive systems. Moreover, the recent work of Godefroid et al. [14] has shown the promise of learning input formats automatically, albeit for a specific format namely PDF. In future, we plan to study this direction to further alleviate the one-time manual effort of specifying an input format. Last but not the least, we can use the flexible architecture of AFLSMART (Figure 6) to support interfacing with many other input-format-aware blackbox fuzzers, such as the Domato fuzzer [33] which is known to work well for HTML format. This will enhance the utility of AFLSMART for a wider variety of file formats.

Acknowledgments

References

[1] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[2] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[3] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, 2016.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, 2008.

[6] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016, pp. 110–121.

[11] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 660–677.

[12] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based white-box fuzzing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[13] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[14] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[15] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[16] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 720–725.

[17] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, vol. abs/1709.07101, 2017.

[18] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[19] Y. Lian and Z. Hu, "Smarter peach: Add eyes to peach fuzzer," https://www.slideshare.net/rootedcon/yihan-lian-zhibin-hu-smarter-peach-add-eyes-to-peach-fuzzer-rooted2017, 2018.

[20] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[21] H. Peng, Y. Shositaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[22] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[4] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2003.

[23] V. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[24] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, no. 12, pp. 366–375, 1972.

[25] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of 24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[26] E. J. Schwartz, T. Avgerinos, and D. Brumley, "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*, ser. SP '10, 2010, pp. 317–331.

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[28] Website, "010editor - hex editor," https://www.sweetscape.com/010editor/, 2018.

[29] ——, "010editor templates," https://www.sweetscape.com/010editor/repository/templates/, 2018.

[30] ——, "Afl dictionary," https://lcamtuf.blogspot.com.au/2015/01/afl-fuzz-making-up-grammar-with.html, 2018.

[31] ——, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018.

[32] ——, "Common vulnerability scoring system v3.0: Specification document," https://www.first.org/cvss/specification-document, 2018.

[33] ——, "Domato: A DOM fuzzer," https://github.com/google/domato, 2018.

[34] ——, "Explanation of the wave file format specification," http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html, 2018.

[35] ——, "Hackernews on afl-fuzz," https://news.ycombinator.com/item?id=9489441, 2018.

[36] ——, "honggfuzz," https://github.com/google/honggfuzz, 2018.

[37] ——, "libFuzzer: A library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, 2018.

[38] ——, "Peach Fuzzer: Discover unknown vulnerabilities," https://www.peach.tech/, 2018.

[39] ——, "SPIKE," http://www.immunitysec.com/downloads/SPIKE2.9.tgz, 2018.

[40] ——, "WavPack: A hybrid lossless audio compression library," http://www.wavpack.com/, 2018.