Smart Greybox Fuzzing

Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa,
Alexandru Răzvan Căciulescu, and Abhik Roychoudhury

Abstract—Coverage-based greybox fuzzing (CGF) is one of the most successful approaches for automated vulnerability detection. Given a seed file (as a sequence of bits), a CGF randomly flips, deletes or copies some 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 complex chunk-based files, our tool AFLSMART achieves substantially more branch coverage (up to 87% improvement) and exposes more vulnerabilities than baseline AFL. Our tool AFLSMART discovered 42 zero-day vulnerabilities in widely-used, well-tested tools and libraries; 22 CVEs were assigned.

Index Terms—vulnerability detection, smart fuzzing, automated testing, file format, grammar, input structure

1 Introduction

Coverage-based greybox fuzzing (CGF) is a popular and effective approach for vulnerability discovery. 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. American Fuzzy Lop (AFL) [39], its variants [3], [4], [11], [20], [21], [28], [34], as well as Libfuzzer [46] constitute the most widely-used implementations 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 bitflips). 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 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 result in invalid files that are 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 [38] and dynamic taint analysis [32]. 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 [43] 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. [32] leverage dynamic taint analysis [33] 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 [19], [47] 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 [47] 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 [19] 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 address the challenge of enabling structural mutation for partially valid seed inputs, i.e., files that do not fully adhere to the provided grammar. We introduce a novel *validity-based power schedule* that assigns more energy to seeds with a higher degree of validity. This schedule 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. AFLSMART works for all complex file formats that follow a tree structure where individual nodes are called data chunks. Such chunk-based formats are *prevalent*, i.e., most common file formats are chunk-based[1] and *important*, i.e., because chunk-based file formats are used as the most popular means to exchange data between machines, they form a common attack vector to compromise software systems.

Our evaluation demonstrates that AFLSMART, within a given time limit of 24 hours, can double the zero-day bugs found. AFLSMART discovers 33 bugs (13 CVEs assigned) while the baseline (AFL and its extension AFLFAST [4]) 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 branch coverage up to 87% compared to the baseline. AFLSMART also outperforms VUZZER [32] 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 [47]) 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 [47]), 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 chunk-based file formats such as AVI 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 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.

[1] https://fileinfo.com/filetypes/common

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

Fig. 1: An excerpt of the WAVE file format (from Ref. [42])

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 [49], 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 [39] or AFLFAST [4].

The discovered vulnerability (CVE-2018-10536) is a *buffer overwrite* in the WAVE-parser component of WavPack. 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).

```
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 ...
```

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

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 6–10 in Figure 2). 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 $S_{✗}$
 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 $S_{✗}$
```

We use these vulnerabilities to illustrate the shortcomings of traditional greybox fuzzing. Algorithm 1, which is extracted from [4], shows the general greybox fuzzing loop. The fuzzer is provided with an 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) [39].

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.

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
...		

Fig. 3: Canonical WAVE file (from Ref. [42])

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. Unlike smart blackbox fuzzing, SGF leverages coverage-information and a power schedule 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.

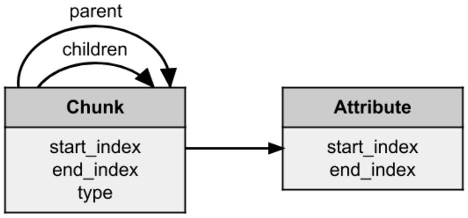


Fig. 4: Virtual structure used by AFLSMART

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>
```

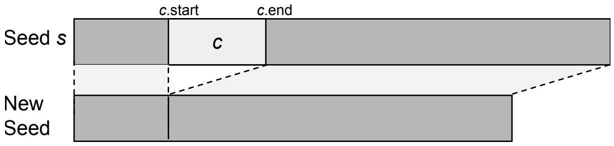
Listing 1: WAVE Peach Pit File Format Specification

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`.

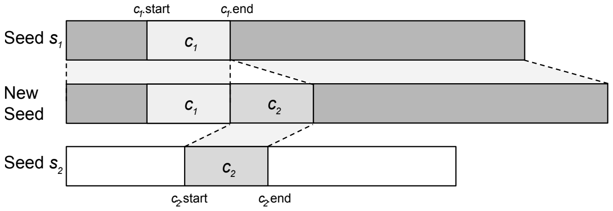
To construct the virtual structure, a file format specification and a parser is required. Given the specification and the file, the parser constructs the virtual structure. For example, Peach [47] has a robust parser component called *File Cracker*. Given an input file and the file format specification, 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

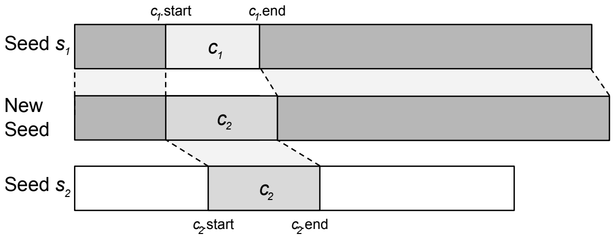
Based on this virtual input structure, we define three generic structural mutation operators – *smart deletion*, *smart addition* and *smart splicing*.



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 structural integrity. A key challenge of existing bit-level mutation operators is to maintain the structural integrity of the generated inputs. This is primarily addressed by structural mutation operators. However, there is no guarantee that our structural mutation operators maintain structural integrity. 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). Moreover, the specification of *immutable attributes* allows the smart greybox fuzzer to apply bit-level mutation operators only to indices of attributes that *are* mutatable. Strictly enforcing the structural integrity is not always desirable while a high degree of validity is necessary to reach beyond the parser code. Our case study demonstrates that this relaxation is a critical advantage of our lightweight virtual structure design.

Maintaining semantic integrity. A second key challenge of any mutational fuzzer is to maintain implicity constraints across data fields. Modifying the bytes in one data field might require an intelligent modification of the bytes in another field, such as the checksum computed over the data field. Smart greybox fuzzing can address this in several ways. Firstly, such implicit contraints are maintained *within* fragments that are inserted. A similar observation was made by Holler et al. [19], [55] while developing LangFuzz, a smart, mutational blackbox fuzzer. Secondly, some constraints such as checksum can be repaired a-posteriori (e.g., using the Peach fixups [51]). However, there is no general solution to repair generated files that are corrupted because of unknown or broken implicit constraints across data fields.

3.3 Region-based 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 would cross chunk-boundaries, the mutation is not applied.[2]

3.3.2 Fragment- and Region-based Mutation

After implementing stacking mutations, we observed that many inputs were added to the seed corpus which are invalid w.r.t. the format specification. AFLSMART used the specification to disassemble a valid file into fragments. A *fragment* is a subtree in the parse tree of a file. These fragments could be added, deleted, and substituted as described in Section 3.2. However, most newly added seeds could not be parsed successfully. Without a successful parsing, there was no parse tree. Our fragment-based smart greybox fuzzing quickly degenerated to a dumb greybox fuzzer.

We addressed this challenge using regions returned via the parser's parse table. A *region* is contiguous sequence of bytes in the file that are associated with a data chunk or an attribute in the specification. If the file is corrupted, the parser will fail at some point. Until this point, regions can be derived that adhere to the specification. To populate our virtual structure, AFLSMART uses the parse table within the Peach Cracker component to derive for each chunk and attribute the start and end index as well as the type.

3.3.3 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* [4]. 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 [4]. 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 [39] assigns more energy to smaller seeds with a lower execution time that have been discovered later. The gradient descent-based power schedule of AFLFAST [4] 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.

[2] The benefit of stacking simple and structural mutations is explored further in the Fuzzing Book [55].

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

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

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 languagues (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>
```

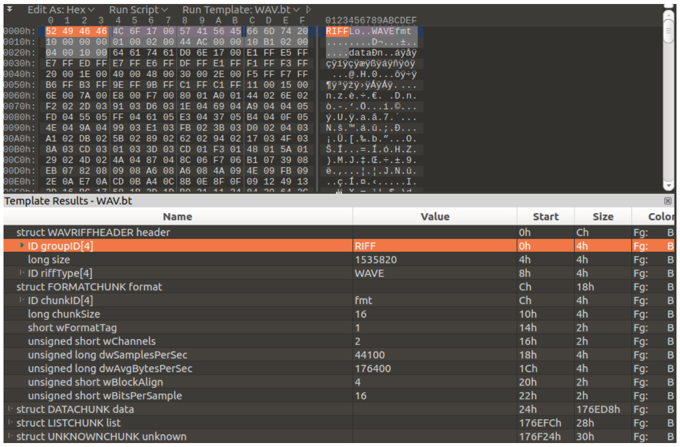
Listing 2: Generic Chunk Model

```
<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 [36] 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) [37].



```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 52 49 46 46 4C 6F 17 00 57 41 56 45 66 6D 74 20  RIFFLo..WAVEfmt 
0010h: 10 00 00 00 01 00 02 00 44 AC 00 00 10 B1 02 00  ........D¬...±..
0020h: 04 00 10 00 64 61 74 61 D0 6E 17 00 E1 FF E5 FF  ....dataÐn..áÿåÿ
0030h: E7 FF ED FF E7 FF E6 FF DF FF E1 FF F1 FF F3 FF  çÿíÿçÿæÿßÿáÿñÿóÿ
0040h: 20 00 1E 00 40 00 48 00 30 00 2E 00 F5 FF F7 FF   ...@.H.0...õÿ÷ÿ
0050h: B6 FF B3 FF 9E FF 9B FF C1 FF C1 FF 11 00 15 00  ¶ÿ³ÿžÿ›ÿÁÿÁÿ....
0060h: 6E 00 7A 00 E8 00 F7 00 80 01 A0 01 44 02 6E 02  n.z.è.÷.€. .D.n.
0070h: F2 02 2D 03 91 03 D6 03 1E 04 69 04 A9 04 04 05  ò.-.‘.Ö...i.©...
0080h: FD 04 55 05 FF 04 61 05 E3 04 37 05 B4 04 0F 05  ý.U.ÿ.a.ã.7.´...
0090h: 4E 04 9A 04 99 03 E1 03 FB 02 3B 03 D0 02 04 03  N.š.™.á.û.;.Ð...
00A0h: A1 02 DB 02 5B 02 89 02 62 02 94 02 17 03 4F 03  ¡.Û.[.‰.b.”...O.
00B0h: 8A 03 CD 03 01 03 3D 03 CD 01 F3 01 48 01 5A 01  Š.Í...=.Í.ó.H.Z.
00C0h: 29 02 4D 02 4A 04 87 04 8C 06 F7 06 B1 07 39 08  ).M.J.‡.Œ.÷.±.9.
00D0h: EB 07 82 08 09 08 A6 08 A6 08 4A 09 4E 09 FB 09  ë.‚...¦.¦.J.N.û.
00E0h: 2E 0A E7 0A CD 0B A4 0C 8B 0E 8F 0F 09 12 49 13  ..ç.Í.¤.‹.....I.
```

Name	Value	Start	Size	Color
struct WAVRIFFHEADER header		0h	Ch	Fg: B
ID groupID[4]	RIFF	0h	4h	Fg: B
long size	1535820	4h	4h	Fg: B
ID riffType[4]	WAVE	8h	4h	Fg: B
struct FORMATCHUNK format		Ch	18h	Fg: B
ID chunkID[4]	fmt	Ch	4h	Fg: B
long chunkSize	16	10h	4h	Fg: B
short wFormatTag	1	14h	2h	Fg: B
unsigned short wChannels	2	16h	2h	Fg: B
unsigned long dwSamplesPerSec	44100	18h	4h	Fg: B
unsigned long dwAvgBytesPerSec	176400	1Ch	4h	Fg: B
unsigned short wBlockAlign	4	20h	2h	Fg: B
unsigned short wBitsPerSample	16	22h	2h	Fg: B
struct DATACHUNK data		24h	176ED8h	Fg: B
struct LISTCHUNK list		176EFCh	28h	Fg: B
struct UNKNOWNCHUNK unknown		176F24h	30h	Fg: B

Fig. 5: Analyzing file structure using 010Editor

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 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[3] 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 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 [39] and AFLFast [4]. 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 [47]. We also compare AFLSmart with VUzzer [32]. 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 discovers more unique bugs than AFL/AFLFast in 24 hours, and in the absence of bugs whether AFLSmart achieves higher branch coverage than AFL/AFLFast in the given time.

RQ-2. *Is smart greybox fuzzing more effective and efficient than smart blackbox fuzzing?* Specifically, we investigate whether AFLSmart discovers more unique bugs than Peach in 24 hours, and in the absence of bugs whether AFLSmart achieves higher branch coverage than Peach in the given time budget.

RQ-3. *Does mutation stacking contribute to the effectiveness of smart greybox fuzzing?* Specifically, we compare the branch coverage achieved by AFLSmart in two settings—with and without stack mutations.

RQ-4. *Is smart greybox fuzzing more effective than taint analysis-based greybox fuzzing?* Specifically, we investigate the number of unique bugs found by Vuzzer and AFLSmart individually and 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.

[3] 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

TABLE 1: Subject Programs (18) and File Formats (10). VUzzer subject programs (6) are at the bottom. At runtime, AFL-based fuzzers replace "@@" by a path to the file containing the mutated data.

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**	`-i @@ -o out.png`
LibJasper	Image processing	33 K	`jasper`	**JP2**	`-f @@ -t jp2 -F /dev/null`
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**	`@@`

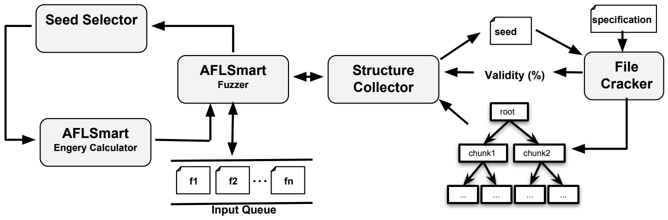


Fig. 6: Architecture of AFLSmart

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) [47] which fully supports highly-structured file formats such as PNG, JPEG, GIF, MP3, WAV and AVI. It is worth noting that we *only* use and modify the File Cracker component of Peach for parsing (i.e., cracking) the seed corpus. AFLSmart does *not* integrate Peach's fuzzing logic or its mutation operators. Our smart mutation operators are designed and implemented on top of AFL.

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 [8] and the instrumented code can be processed by AFL. Such a binary fuzzing approach has been achieved in the WinAFL tool[4] 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 [14] 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 six (6) chunk-based 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[5], LibPNG[6], LibJpeg-Turbo[7], ImageMagick[8], and OpenJPEG[9])

[4] https://github.com/ivanfratric/winafl
[5] https://github.com/FFmpeg/FFmpeg
[6] https://github.com/glennrp/libpng
[7] https://github.com/libjpeg-turbo
[8] https://github.com/ImageMagick/ImageMagick
[9] https://github.com/uclouvain/openjpeg

have joined the Google OSS-Fuzz project[10] and they are continuously tested using the state-of-the-art fuzzers including AFL and LibFuzzer. LibAV[11], WavPack [12] and Libjasper[13] are widely-used libraries and tools for image, audio and video files processing and streaming. Binutils[14] is a set of utilities for analyzing binary executable files. It is installed on almost all Linux-based machines.

To compare with VUzzer (RQ-4), 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), tcpdump (v4.5.1), tcptrace (v6.6.7), and djpeg (v1.3.0). These subjects take MP3, GIF, PDF, PCAP and JPEG files as inputs. At the time we conducted our experiments, VUzzer had not supported 64-bit environment.

Table 1 shows the full list of programs and their information. Note that the sizes of subject programs are calculated by `sloccount`[15]. Moreover, to increase the reproducibility of our experiments, in the fifth column 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.[16] 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[17], 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 inputs from websites keeping sample files (WAV[18], AVI[19], JP2[20], PCAP[21], MP3[22], GIF[23] and PDF[24]). 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. AFLSmart, AFL, and AFLFast were run with the same dictionaries.

TABLE 2: File Format Specifications and Seed Corpora

File Format Specification			Seed Corpus	
Format	Length (#Lines)	Time spent	#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

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 had the same time budget (24 hours), the same computational resources, and were started with the same seed corpus with the same dictionaries. Peach and AFLSmart also used the same Peach Pits (i.e., grammars). In the comparison with VUzzer, 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. In this environment, both VUzzer and AFLSmart were started with the same seed corpus.

Experiment repetition. To mitigate the impact of randomness, for each subject program we ran 20 isolated instances of each of AFL, AFLFast, AFLSmart, and Peach. We emphasize that *none* of the instances shared the same queue.[25] Specifically, Peach does not support such a shared queue architecture.

Settings for AFL and AFLFast. We ran AFL with option "-d" to enable its Fidgety mode which significantly boosts its efficiency (as explained by the creator of AFL).[26] The FidgetyAFL was a result of investigating the power schedules designed in AFLFast. For AFLFast, we ran its default setting which uses the COE power schedule.

Measuring branch coverage. To calculate branch coverage, we used the `gcov`-tool. Unlike AFL-based fuzzers, Peach does not keep any generated test cases. It only stores bug-triggering inputs. So we modified Peach such that it stores all test cases which Peach generates during a 24-hour run.

Measuring #unique bugs. To calculate the number of unique bugs found by a technique, we started with an automatic call-stack-based bucketing approach [13]: Crashes that have the same call stack are in the same group. We then manually analyzed the resulting groups, and selected one representative from each group for bug reporting purposes.

[10] https://github.com/google/oss-fuzz
[11] https://github.com/libav/libav
[12] https://github.com/dbry/WavPack
[13] https://github.com/mdadams/jasper
[14] https://www.gnu.org/software/binutils/
[15] https://www.dwheeler.com/sloccount/
[16] http://community.peachfuzzer.com/v3/PeachPit.html
[17] http://community.peachfuzzer.com/v3/TutorialFileFuzzing/
[18] https://freewavesamples.com/source/roland-jv-2080
[19] http://www.engr.colostate.edu/me/facil/dynamics/avis.htm
[20] http://samples.ffmpeg.org/
[21] https://wiki.wireshark.org/SampleCaptures
[22] https://www.magnac.com/sounds.shtml
[23] https://people.sc.fsu.edu/ jburkardt/data/gif/gif.html
[24] https://www.pdfa.org/isartor-test-suite/
[25] https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt
[26] https://groups.google.com/forum/#!topic/afl-users/1PmKJC-EKZ0

TABLE 3: Average branch coverage, coverage factor w.r.t. AFL, Vargha-Delaney effect size A^{12} w.r.t. AFL (statistically significant effect sizes in **bold**; using Wilcoxon signed-rank test), and number of unique bugs discovered in 20 runs with a 24 hours time budget. Each unique bug has its own bug-id.

Binary	Fuzzer	Coverage	Factor	A^{12}	#Bugs
`readelf`	AFL	49.51%	100%	-	3
ELF	AFLFAST	46.82%	95%	**0.16**	3
	Peach	25.57%	52%	**0.00**	0
	AFLSMART	48.07%	97%	**0.26**	3
`nm-new`	AFL	14.04%	100%	-	1
ELF	AFLFAST	13.68%	97%	0.42	1
	Peach	8.02%	57%	**0.00**	0
	AFLSMART	14.30%	102%	0.60	2
`pngimage`	AFL	40.02%	100%	-	0
PNG	AFLFAST	39.80%	99%	0.37	0
	Peach	26.86%	67%	**0.00**	0
	AFLSMART	40.39%	101%	**0.70**	1
`magick`	AFL	3.34%	100%	-	0
PNG	AFLFAST	3.16%	95%	**0.27**	0
	Peach	2.80%	84%	**0.00**	0
	AFLSMART	3.27%	98%	0.41	0
`djpeg`	AFL	19.83%	100%	-	0
JPEG	AFLFAST	19.97%	101%	0.50	0
	Peach	10.55%	53%	**0.00**	0
	AFLSMART	19.96%	101%	0.48	0
`imginfo`	AFL	14.81%	100%	-	2
JPEG	AFLFAST	14.77%	100%	0.50	2
	Peach	1.44%	10%	**0.00**	0
	AFLSMART	14.43%	97%	0.39	2
`ffmpeg`	AFL	3.94%	100%	-	0
AVI	AFLFAST	3.91%	99%	0.41	0
	Peach	4.22%	107%	**0.98**	0
	AFLSMART	5.96%	151%	**1.00**	1
`avconv`	AFL	4.58%	100%	-	3
AVI	AFLFAST	4.68%	102%	0.62	3
	Peach	4.05%	88%	**0.00**	0
	AFLSMART	8.56%	187%	**1.00**	3
`avconv`	AFL	5.97%	100%	-	0
WAV	AFLFAST	5.93%	99%	0.48	0
	Peach	5.24%	88%	**0.06**	0
	AFLSMART	7.08%	119%	**0.84**	3
`wavpack`	AFL	14.40%	100%	-	1
WAV	AFLFAST	14.72%	103%	0.57	1
	Peach	14.62%	102%	**0.27**	1
	AFLSMART	16.36%	114%	**1.00**	5
`decompress`	AFL	47.84%	100%	-	0
JPEG2000	AFLFAST	47.79%	100%	0.54	0
	Peach	25.02%	52%	**0.00**	0
	AFLSMART	47.91%	100%	0.50	3
`jasper`	AFL	27.45%	100%	-	6
JPEG2000	AFLFAST	27.32%	99%	0.47	7
	Peach	19.80%	72%	**0.00**	0
	AFLSMART	29.22%	106%	**0.89**	10

TABLE 4: Statistics on bugs found in 20 runs. ✗ - no bug found. N/20 - the bug was discovered in N out of 20 runs.

Subject	Bug-ID	AFL	AFLFAST	Peach	AFLSMART
WavPack	CVE-2018-10536	✗	✗	✗	20/20
	CVE-2018-10537	✗	✗	✗	12/20
	CVE-2018-10538	✗	✗	✗	20/20
	CVE-2018-10539	✗	✗	✗	15/20
	CVE-2018-10540	10/20	15/20	11/20	12/20
Binutils	Bugzilla-23062	10/20	11/20	✗	11/20
	Bugzilla-23063	13/20	12/20	✗	10/20
	CVE-2018-10372	16/20	18/20	✗	16/20
	CVE-2018-10373	11/20	12/20	✗	14/20
	Bugzilla-23177	✗	✗	✗	13/20
LibPNG	CVE-2018-13785	✗	✗	✗	6/20
Libjasper	Issue-174	8/20	9/20	✗	9/20
	Issue-175	12/20	14/20	✗	12/20
	CVE-2018-19539	✗	✗	✗	15/20
	CVE-2018-19540	✗	✗	✗	7/20
	CVE-2018-19541	✗	✗	✗	6/20
	CVE-2018-19542	✗	7/20	✗	9/20
	CVE-2018-19543	8/20	12/20	✗	13/20
	Issue-182-6	19/20	20/20	✗	18/20
	Issue-182-7	16/20	18/20	✗	19/20
	Issue-182-8	12/20	13/20	✗	16/20
	Issue-182-9	12/20	14/20	✗	11/20
	Issue-182-10	14/20	11/20	✗	15/20
OpenJPEG	Email-Report-1	✗	✗	✗	8/20
	Email-Report-2	✗	✗	✗	13/20
	Issue-1125	✗	✗	✗	15/20
LibAV	Bugzilla-1121	✗	✗	✗	5/20
	Bugzilla-1122	✗	✗	✗	6/20
	Bugzilla-1123	18/20	18/20	✗	18/20
	Bugzilla-1124	15/20	18/20	✗	16/20
	Bugzilla-1125	✗	✗	✗	8/20
	Bugzilla-1127	13/20	15/20	✗	18/20
FFmpeg	Email-Report-3	✗	✗	✗	3/20

6 EXPERIMENTAL RESULTS

RQ.1 SGF Versus Traditional Greybox Fuzzing

In terms of branch coverage, AFLSMART *clearly outperforms both AFL and* AFLFAST (Table 3). On average, AFLSMART achieved 14.40% more branch coverage than AFL which is the second best fuzzer in our experiments. Specifically, AFLSMART covered more branches in nine (9) out of twelve (12) subjects. AFLSMART performed particularly well for the complex file formats (video and audio files) of the two larger subjects, ffmpeg and avconv; AFLSMART explored 51.02% and 86.90% more branches, respectively. Figure 7 explains this significant improvement using an important internal statistic of all AFL-based fuzzers – the number of paths[27] discovered over time. In ffmpeg, avconv-avi, and avconv-wav AFLSMART discovered 250%, 293% and 100% more paths than AFL. AFLSMART performed slightly worse than AFL in a ELF-parsing subject in Binutils (readelf) and the results are on par on magick (ImageMagick utilities) and imginfo (Jasper library). We believe there are two reasons. First, AFL is already known to perform well for binary formats, such as ELF. Secondly, these format require semantic constraints to be satisfied over the input that span more than one data chunk, such as offset-definitions.

Table 3 reports two measures of effect size and one measure of statistical significance (marked in bold) as recommended by Arcuri et al. [1]. *Factor* gives the coverage of the competing technique as a factor of the coverage of AFL (higher is better). *Vargha-Delaney* A^{12} gives the probability that one run of the competing technique is better than one run of AFL. Values below 0.5 indicate that AFL is better while values above 0.5 indicate that the competing technique is better. The *Wilcoxon signed rank test* is used to test whether the effect size is statistically significant.

In terms of bug finding, AFLSMART *discovered bugs in 10 subjects while AFL and* AFLFAST *could not detect bug in four of them* (Tables 3 & 4). After analyzing the crashes, we reported 33 zero-day bugs found by AFLSMART out of which

[27] In AFL and other fuzzers built on top of it, number of paths is number of interesting seeds retained in the queue

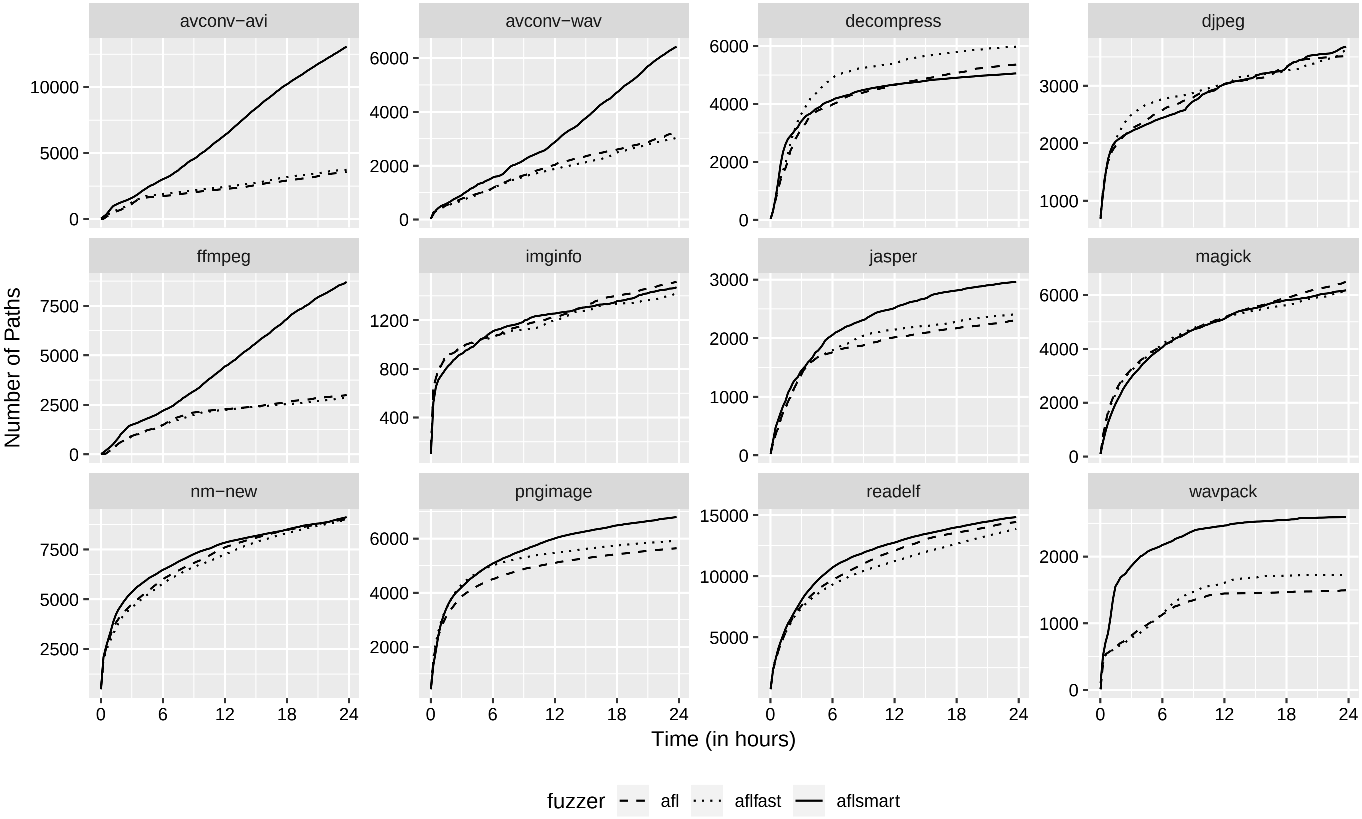


Fig. 7: Number of paths discovered over time for AFL, AFLFAST, and AFLSMART (average of 20 runs).

only 17 bugs were also found by AFL and AFLFAST. Vice versa, all zero-day bugs that AFL and AFLFAST found were also found by AFLSMART. Hence, AFLSMART discovered almost 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 17 bugs we reported. The MITRE corporation[28] has assigned 13 CVEs to the most critical vulnerabilities. In Table 4, for each unique bug we also report the number of runs (out of 20 runs) a technique had discovered the bug.

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` $\geq$ 0x80000000.

```
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,..)) {
```

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

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 (Email-Report-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

[28] https://cve.mitre.org/

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.

```
r = image->comps[0].data;
g = image->comps[1].data;
b = image->comps[2].data;
...
for (i = 0U; i < max; ++i) {
  *in++ = (unsigned char) * r++;
  *in++ = (unsigned char) * g++;
  *in++ = (unsigned char) * b++;
}
...
cmsDoTransform(transform, inbuf, outbuf, ...);
...
r = image->comps[0].data;
g = image->comps[1].data;
b = image->comps[2].data;
...
for (i = 0U; i < max; ++i) {
  *r++ = (unsigned char) * out++;
  *g++ = (unsigned char) * out++;
  *b++ = (unsigned char) * out++;
}
```

Fig. 9: Showing `common/color.c` @ revision `d2205ba`

RQ.2 SGF Versus Smart Blackbox Fuzzing

Given the same input format specifications, AFLSmart *clearly outperforms Peach in all twelve (12) subjects* (see Tables 3 & 4). AFLSmart improved the branch coverage by 133.95% on average and discovered 33 zero-day bugs while Peach could find only one vulnerability in the WavPack library.

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 [22] 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.

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 Contribution of Stack Mutations

In 9 out of 12 subjects, AFLSmart *with stacking optimization outperforms* AFLSmart *without stacking optimization (*AFLSmart**)* (Table 5). To determine the contribution of the stacking optimization (Sec. 3.3.1), we ran AFLSmart with two settings, one where stacking is enabled (AFLSmart) and one where it is disabled (AFLSmart*). Table 5 shows the average branch coverage (in 20 runs). The results indicate that stacking mutations does contribute to the effectiveness of AFLSmart.

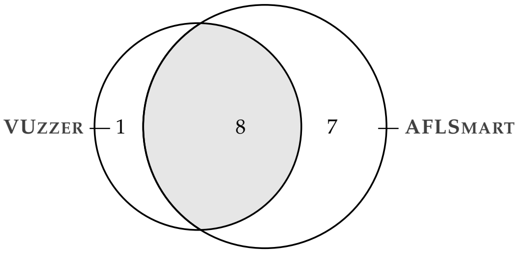


Fig. 10: Venn diagram. Number of bugs that VUzzer and AFLSmart discover individually and together.

RQ.4 SGF Versus Taint-Based Greybox Fuzzing

AFLSmart *outperforms* VUzzer [32] *on* VUzzer*'s benchmark programs*. 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 6). 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; 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 discovered 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 Finding With 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).

TABLE 5: Average branch coverage (in 20 runs) achieved by AFLSmart with stack mutations optimization (AFLSmart) and AFLSmart without the optimization (AFLSmart*)

	readelf	nm-new	pngimage	magick	djpeg	imginfo	ffmpeg	avconv-avi	avconv-wav	wavpack	decompress	jasper
AFLSmart*	47.61%	14.11%	37.49%	**3.29%**	19.73%	**14.60%**	**6.18%**	6.68%	8.06%	14.98%	46.28%	28.61%
AFLSmart	**48.07%**	**14.30%**	**40.39%**	3.27%	**19.96%**	14.42%	5.95%	**7.08%**	**8.56%**	**16.35%**	**47.91%**	**29.22%**

TABLE 6: VUzzer vs AFLSmart on VUzzer's benchmark

Application	VUzzer	AFLSmart
mpg321	2	2
gif2png+libpng	1	2
pdf2svg+libpoppler	3	2
tcpdump+libpcap	1	6
tcptrace+libpcap	1	2
djpeg+libjpeg	1	1

TABLE 7: 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

We run five (5) instances of AFLSmart in parallel mode[29] 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 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 7 shows the CVEs and their severity levels based on the Common Vulnerability Scoring System version 3.0 [40]; all these nine vulnerabilities are rated from medium to high severity.

The results confirm the practical impact of smart greybox 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

Fuzzing is a fast-growing research topic, and making greybox fuzzing grammar-aware has been a natural next step. Since submitting the first draft of the present article, we have become aware of several concurrent research efforts. In the following, we discuss this stream of concurrent works and how *smart greybox fuzzing* as implemented in AFLSmart is different from those. If the reader is keen to try out the various approaches to grammar-based greybox fuzzing, we refer to the chapter "Greybox Fuzzing with Grammars" in the Fuzzingbook [55], a hands-on, tutorial-style textbook on fuzzing with executable examples. For a more general discussion, we refer to the excellent survey of recent advances in fuzzing from Manés et al. [24].

LangFuzz [19] is a fragment-based mutational blackbox fuzzer. Given a context-free grammar and a seed corpus, LangFuzz would first disassemble each seed input into fragments. A *fragment* is a subtree in a seed's parse tree. It is typed by the grammar symbol of subtree's root node. The *fragment pool* is the set of derived fragments. Now, LangFuzz generates new inputs by manipulating existing fragments in a given seed: A fragment may be deleted or substituted by another fragment of the same type. The main advantage is that implicit constraints, such as checksums, are maintained within "real-world" fragments. Other smart blackbox fuzzers include Peach [47], Spike [48], and Domato [41].

Superion [35] conceptually extends LangFuzz with coverage-feedback: Structurally mutated seeds that increase coverage are retained for further fuzzing. In contrast to AFLSmart, Superion does *not* implement deferred parsing or leverage byte-level mutation. The constrained nature of the mutation operators in Superion constrains the set of inputs that can be generated (as compared to AFLSmart which works with a bigger search space). In other words, AFLSmart generates (slightly) invalid inputs to discover bugs in the parser and to achieve more coverage faster [55].

Nautilus [2] integrates fragment-based and byte-level mutational greybox fuzzing. It maintains the parse tree for all seeds and (unlike AFLSmart) for all generated inputs. To allow AFL-style byte-level mutations, it "collapses" subtrees back to byte-level representations. This has the advantage that generated seeds do not need to be re-parsed. However, we believe that over time Nautilus de-generates to structure-unaware greybox fuzzing. Collapsed subtrees are never re-parsed. So, there is a chance that parse-trees of seeds, which are added in a late stage of the fuzzing campaign, are collapsed entirely. In contrast, AFLSmart re-parses each generated input that is added to the queue. To keep the parsing overhead at bay, we introduce deferred parsing. In contrast to Nautilus, AFLSmart also features region-based fuzzing and a validity-based power schedule when the seed input is valid only to some degree.

ProFuzzer [53], SLF [54], and PDF [25] implement region-based fuzzing *without* a grammar. They identify contigiuous regions by incrementally mutating input bytes and observing the changes in coverage.[30] Once the input fields are identified and classified, ProFuzzer applies field-aware mutations such as mutating the whole field instead of individual bytes (e.g., for magic numbers) and updating input data accordingly to satisfy the fields' constraints (e.g., size-of, offset-of). Moreover, ProFuzzer ignores the raw data which could not lead to any new code coverage improvement. While ProFuzzer requires a valid seed corpus, SLF and PDF go one step further by generating valid seeds "out of thin air". They incrementally identify data fields by detecting and satisfying input checks in the parser. In

[29] https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

[30] ProFuzzer and SLF took inspiration from `afl-analyze`, a tool in the AFL toolset that identifies contiguous regions in a similar fashion.

contrast, AFLSMART understands the high-level structure of seed files. ProFuzzer, SLF, and PDF can indeed identify contiguous regions in a file, but they cannot determine the type of these regions (e.g., IHDR in a PNG file) or coarser structures of regions (i.e., fragments).

LibProtobuf [50] and Zest [26], [27] introduce smart greybox fuzzing to the unit level, i.e., for specific program methods. LibProtobuf compiles a grammar-specification into a fuzzer driver stub for the coverage-based greybox fuzzer, LibFuzzer [46]. This fuzzer driver translates byte-level mutations of LibFuzzer into structural mutations of the fuzzer target. However, the fuzzer driver still needs to be manually wired to the fuzzer target (e.g., the XML-parser function of LibXML). Now, Zest integrates coverage- and property-based testing and implements a coverage-guided parameter search over the input variables of a fuzzed method. This allows Zest to map mutations in the untyped parameter domain to structural mutations in the input domain. However, while Zest and LibProtobuf focus on the unit level, AFLSMART tackles smart system-level greybox 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 [9] or S^2E [12]. Grammar-based whitebox fuzzers [16] can generate files that are valid w.r.t. a context-free grammar. Model-based whitebox fuzzing [30] 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) [39], [46], which is a popular and effective approach for software vulnerability detection. The AFL fuzzer [39] and its extensions [3], [4], [11], [15], [20], [21], [28], [34] 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.

Boosted greybox fuzzing. AFLFAST [4] 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 [3] performs reachability analysis to a given location or target by prioritizing seeds which are estimated to have a lower distance to the target. Angora [11] is an extension of AFL to improve its coverage that performs search based on gradient descent to solve path condition without symbolic execution. SlowFuzz [29] 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 [32] 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 [21] 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 [10] 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. While greybox fuzzing can generate tens of thousands of inputs per second symbolic execution can systematically explore the behaviors of the system. How to integrate both techniques effectively is an active research topic [7]. T-Fuzz [28] removes sanity checks in the code that blocks the fuzzers (AFL or honggfuzz [44]) from progressing further. This, however, introduces false positives, which are then detected using symbolic execution. Driller [34] 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 [23] present an approach to derive the file's input tree from the dynamic execution trace. Learn&Fuzz [18] 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 [17]. Such works on input format inference can potentially help input-aware fuzzers such as AFLSMART.

9 DISCUSSION

Smart fuzzers needed. 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 87%) higher branch 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.

Real-world impact. Our work on file-format aware greybox fuzzing has generated significant interest both from industry and media. After our work was made available openly via Arxiv [31], we were reached out to by the lib-protobuf team [50] at Google–for exploring the industrial use of our smart fuzzing technologies. Subsequent to these discussions between us and the lib-protobuf team, the lib-protobuf team has also shared some reflections on smart fuzzing in a blog [52]. Furthermore, as an ongoing collaboration, we are also making our smart fuzzing technology available to the lib-protobuf team by providing conversion between our file format specifications and lib-protobuf descriptions. Last but not the least, our work has been featured in technology oriented media reports [45] subsequent to our making it available in the public domain via Arxiv [31].

Reproducibility. To ensure the reproducibility of our experiments, we have made AFLSMART open source at

https://github.com/aflsmart/aflsmart.

The Github respository contains the source code of AFLSMART, as well as the seed corpora, dictionaries, and Peach Pits (i.e., grammars) that we used in our experiments. Moreover, we ported the underlying algorithms and optimizations to Python for everyone to try and experiment with. The executable Python code is presented and explained in a tutorial-style book chapter in the Fuzzing Book [55].

Future work. 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. [18] has shown the promise of learning input formats automatically, albeit for a specific format namely PDF. We plan to study this direction to further alleviate the one-time manual effort of specifying an input format. Another research direction is the provision of assurances about the automated vulnerability discovery process [5], [6].

ACKNOWLEDGMENTS

This research was partially supported by a grant from the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCRNCR001-21) and administered by the National Cybersecurity R&D Directorate. This research was partially funded by the Australian Government through an Australian Research Council Discovery Early Career Researcher Award (DE190100046).

REFERENCES

[1] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, May 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] 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.

[4] M. Böhme, V.-T. 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, pp. 1032–1043.

[5] M. Böhme, "STADS: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 2, pp. 7:1–7:52, Jun. 2018.

[6] ——, "Assurances in software testing: A roadmap," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE 2019, 2019, pp. 1–4.

[7] 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, April 2016.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 660–677. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00040

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

[17] 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.

[18] 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.

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

[20] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[21] 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.

[22] Y. Lian and Z. Hu, "Smarter peach: Add eyes to peach fuzzer," in *RootedCon*, 2017.

[23] 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.

[24] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," 2018.

[25] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, 2019, pp. 548–560.

[26] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Semantic fuzzing with zest," in *ACM Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[27] ——, "Validity fuzzing and parametric generators for effective random testing," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19, 2019, pp. 266–267.

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

[29] 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.

[30] 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.

[31] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," 2018.

[32] 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.

[33] 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. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.26

[34] 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.

[35] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 724–735.

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

[37] ——, "010editor templates," https://www.sweetscape.com/010editor/repository/templates/, 2018.

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

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

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

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

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

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

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

[45] ——, "Hot fuzz: Bug detectives whip up smarter version of classic afl fuzzer to hunt code vulnerabilities," https://www.theregister.co.uk/2018/11/28/better_fuzzer_aflsmart/, 2018.

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

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

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

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

[50] ——, "libprotobuf," https://github.com/google/libprotobuf-mutator, 2019.

[51] ——, "Peach fuzzer: Fixup," https://community.peachfuzzer.com/v3/Fixup.html, 2019.

[52] ——, "Structure-aware fuzzing with libfuzzer," https://github.com/google/fuzzer-test-suite/blob/master/tutorial/structure-aware-fuzzing.md, 2019.

[53] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 882–899.

[54] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: Fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 712–723.

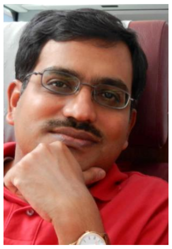
[55] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Greybox fuzzing with grammars," in *Generating Software Tests*. Saarland University, 2019, retrieved 2019-05-21 20:58:06+02:00. [Online]. Available: https://www.fuzzingbook.org/

Van-Thuan Pham is a postdoctoral research fellow at Monash University, Australia. During his PhD studies at NUS, under the supervision of Prof Abhik Roychoudhury he conducted research on fuzz testing techniques (including black-box, coverage-based grey-box and symbolic-execution based white-box fuzzing) and applied these techniques to vulnerability detection, crash reproduction and debugging.

Marcel Böhme is a 2019 ARC DECRA Fellow and lecturer at Monash University, Australia. He was research fellow at CISPA, Saarland University, Germany from 2014 to 2015 and completed his PhD at National University of Singapore in 2014. Marcel's research is focussed on automated vulnerability detection, analysis, testing, debugging, and repair of large software systems. His tools discovered 100+ bugs in widely-used software systems, more than 40 of which are security-critical vulnerabilities registered as CVEs at the US National Vulnerability Database.

Andrew E. Santosa obtained his B.Eng. and M.Eng. degrees from the University of Electro-Communications in 1997 and 1999, respectively. He obtained his Ph.D. degree from the National University of Singapore. He is interested in software analysis and engineering, and he has served in both academia and industry.

Alexandru Răzvan Căciulescu is a Linux and security enthusiast who spends most of his time in Sublime and vim when he isn't slaying 'features' in GDB. He completed his Masters degree at University Politehnica of Bucharest, Romania.



Abhik Roychoudhury is a Professor of Computer Science at National University of Singapore. His research focuses on software testing and analysis, trust-worthy software construction and software security. He is currently leading the Singapore Cyber-security Consortium. He has served as an Associate Editor of IEEE Transactions on Software Engineering (TSE)during 2014-18, and is serving as an Associate Editor of IEEE Transactions on Dependable and Secure Computing (TDSC) during 2019-21. Abhik received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 2000.