FuzzInMem: Fuzzing Programs via In-memory Structures

Xuwei Liu
Purdue University
West Lafayette, United States
liu2598@purdue.edu

Wei You
Renmin University of China
Beijing, China
youwei@ruc.edu.cn

Yepeng Ye
Purdue University
West Lafayette, United States
ye203@purdue.edu

Zhuo Zhang
Purdue University
West Lafayette, United States
zhan3299@purdue.edu

Jianjun Huang*
Renmin University of China
Beijing, China
hjj@ruc.edu.cn

Xiangyu Zhang
Purdue University
West Lafayette, United States
xyzhang@purdue.edu

ABSTRACT

In recent years, coverage-based greybox fuzzing has proven to be an effective and practical technique for discovering software vulnerabilities. The availability of American Fuzzy Loop (AFL) has facilitated numerous advances in overcoming challenges in fuzzing. However, the issue of mutating complex file formats, such as PDF, remains unresolved due to strict constraints. Existing fuzzers often produce mutants that fail to parse by applications, limited by bit/byte mutations performed on input files. Our observation is that most in-memory representations of file formats are simple, and well-designed applications have built-in printer functions to emit these structures as files. Thus, we propose a new technique that mutates the in-memory structures of inputs and utilizes printer functions to regenerate mutated files. Unlike prior approaches that require complex analysis to learn file format constraints, our technique leverages the printer function to preserve format constraints. We implement a prototype called FuzzInMem and compare it with AFL as well as other state-of-the-art fuzzers, including AFL++, Mopt, Weizz, and FormatFuzzer. The results show that FuzzInMem is scalable and substantially outperforms general-purpose fuzzers in terms of valid seed generation and path coverage. By applying FuzzInMem to real-world applications, we found 29 unique vulnerabilities and were awarded 5 CVEs.

CCS CONCEPTS

• **Software and its engineering**; • **Security and privacy → Software and application security**;

KEYWORDS

Fuzzing, Software testing, Program synthesis

ACM Reference Format:
Xuwei Liu, Wei You, Yepeng Ye, Zhuo Zhang, Jianjun Huang, and Xiangyu Zhang. 2024. FuzzInMem: Fuzzing Programs via In-memory Structures . In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639172

*Corresponding author.

1 INTRODUCTION

Coverage-based Greybox Fuzzing is a widely-used software testing technique for vulnerability discovery. It has proven to be one of the most effective security analysis approaches, leading to the identification of many high-impact security flaws. A critical step for fuzzing is the generation of inputs that can cover various program paths and increase the chance of exposing security vulnerabilities. Modern fuzzers typically address this problem by either generating seed inputs following a specific input model or mutating a set of initial seed inputs. However, there are a few hard challenges, even for the state-of-the-art fuzzers.

Generation-based fuzzers [1, 6] rely on an input model describing the input format to produce legitimate test cases. Although the input model can sometimes be recovered from execution traces or program source codes by reverse engineering techniques, in most cases, fuzzers require experts with domain knowledge to provide input specifications. However, the model may be too complex to be complete and lack semantic constraints dictating relations across multiple input fields. As a result, they may not lead to good path coverage. In addition, inputs exploiting vulnerabilities are usually ill-formed files that do not follow input models. Generation-based fuzzers lack the ability to generate such ill-formed seed inputs.

A more popular way is mutating existing seed inputs at the bit and byte level to generate variants that allow exploring new program regions [3, 5]. While this approach works well for simple and less constrained inputs, it may not be effective for inputs with complex structure and/or intensive cross-field constraints. These fuzzers tend to spend much time generating numerous inputs which however are rejected in the initial parsing stage, resulting in little code coverage improvement. Researchers mitigate this problem by adopting different seed scheduling [14, 21, 42, 51], energy allocation strategies [11, 32, 55], and coverage feedback [10, 25]. Hybrid fuzzers [48, 56] combining symbolic execution and mutation-based fuzzing help reach deep states of a program and explore more program paths. However, their efficacy heavily depends on how well the symbolic execution component handles the path conditions in the target program. For example, they may struggle in dealing with length and offset constraints [30, 53].

We argue that input generation is essentially a search within an intrinsically constrained input space starting from some initial input. The objective is to mutate the input as much as possible while keeping input validity. In other words, a good mutant should pass the parsing stage of a program (low parsing errors) but vary from other seeds after parsing (better variation). Mutation-based fuzzers can hardly satisfy both conditions because the low-level bit/byte modification either fails in parsing or incurs insufficient mutations. Hybrid fuzzers can generate interesting seeds but may not scale well in some cases. As a result, an efficient mutation technique that can explore deep program states is desired.

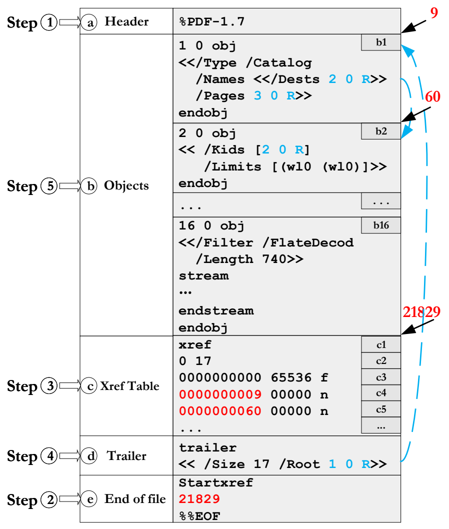
In this paper, we propose a novel technique that mutates in-memory data structures corresponding to inputs and reuses the target program logic to emit the mutated structures back to files. We develop an algorithm to analyze the application to find the key data structures, by locating parser functions and then data structures accessed by those functions. A synthesizer is then leveraged to analyze the source code and synthesize code to automatically mutate the in-memory structures following a number of strategies. In particular, new header files are generated to redefine the key data structures. The re-definitions are compatible with the original definitions but individual field types may be changed to facilitate mutation. For example, some key fields such as callback function pointers and constants are set to immutable as mutations may completely break their semantics. Mutation functions are also generated to operate based on the re-defined data structures. At runtime, our tool monitors and intercepts the subject program's execution after it finishes parsing and populates the key data structures. It then casts the key data structure instances to their re-defined types and passes them to the mutation functions, which mutate them in memory, e.g., by changing, deleting, inserting, and reusing fields. We further observe that many popular applications have built-in printer functions (see Section 4.1) that export in-memory structures while constructing format constraints on the way (e.g., setting data length by counting the number of data items in memory), from the least-constrained in-memory data. We hence reuse the printer functions in applications to dump the mutated in-memory structures into data files that satisfy format constraints. The emitted data files are hence used in further fuzzing.

Our contributions are summarized as follows:

- We developed a novel technique for mutation-based fuzzing. Starting from a seed input, it parses the input, modifies the in-memory structures, and reuses the printer function to emit valid data files that serve as high-quality seeds.
- We developed a technique that automatically synthesizes data structure definitions and corresponding mutation functions based on the original definitions of key data structures.
- We developed a prototype called FuzzInMem, evaluated it on 15 real-world applications and compared it with state-of-the-art fuzzers including AFL, AFL++, Mopt, Weizz and FormatFuzzer. Our results show 42%-70% improvements in path coverage and 10x-10000x in valid seed generation.
- We found 29 new unique vulnerabilities and 5 exploitable CVEs by applying FuzzInMem on real-world applications.

2 MOTIVATION

We use PDF format as an example to explain the limitations of existing techniques and motivate the idea of FuzzInMem.



```
%PDF-1.7
1 0 obj
<</Type /Catalog
  /Names <</Dests 2 0 R>>
  /Pages 3 0 R>>
endobj
2 0 obj
<< /Kids [2 0 R]
   /Limits [(wl0 (wl0)]>>
endobj
...
16 0 obj
<</Filter /FlateDecod
  /Length 740>>
stream
...
endstream
endobj
xref
0 17
0000000000 65536 f
0000000009 00000 n
0000000060 00000 n
...
trailer
<< /Size 17 /Root 1 0 R>>
Startxref
21829
%%EOF
```

Figure 1: The structure of a PDF file

2.1 PDF and PDF Reader

Portable Document Format (PDF) [52] is one of the most popular formats to represent documents in a manner independent of application software, hardware, and operating systems. Based on the PostScript language, a PDF file encapsulates a complete description of a document in a fixed layout, including text, fonts, vector graphics, raster images, and meta information needed to display it.

The PDF structure as in Figure 1 consists of several parts: header, body, cross-reference table (xref table), trailer, and end-of-file indicator. The header ⓐ is the first line of a PDF file that specifies the version of PDF specification. The body section ⓑ holds all the document's objects being shown to the user, including texts, images, etc. There are in total 9 different data types for objects, among which 6 are primitive types (`null`, `booleans`, `numbers`, `names`, `strings`, and `indirect references`) and others are composite types (`arrays`, `dictionaries`, `streams`). Dictionary and stream are the most common objects in PDF files. A dictionary is a composite type holding name-value pairs. For example, object $\boxed{\text{b1}}$ (the second grey box from the top) is a dictionary with three name-value pairs denoting three properties. Specifically, the first property has name "`/TYPE`" and value "`/Catalog`" indicating that it is a special type of dictionary called *catalog dictionary*; the second property is a sub-dictionary called "`/Names`" that supports name look-up; and the third is an indirect reference called "`/Page`" that points to the third object in the body section denoting a page tree representing pages in the document. A stream, e.g., object $\boxed{\text{b16}}$ consists of text content (delimited by `stream` and `endstream`) and a description dictionary that indicates the stream length (e.g., 740) and optional encryption algorithms (e.g., the "`FlateDecod`" algorithm). Objects refer to others by indirect references (highlighted in blue). The xref table ⓒ records the offsets (highlighted in red) and status of all the objects in the document, which allows random access to objects in the PDF. Each object has an entry in the xref table, which is always 20 bytes long. The PDF trailer ⓓ points out this PDF's root object (from which the whole rendering procedure starts) by an indirect reference to the catalog dictionary. The end-of-file indicator ⓔ specifies the offset pointing to the first item in the xref table (e.g., 21829).

Poppler [7] is a cross-platform open-source PDF library supporting various PDF operations and we show how it parses each segment of a valid PDF file. When the file presented in the right of Figure 1 is passed to Poppler, the program first checks the header ⓐ to ensure that version 1.7 is valid (Step ①). Then it searches for the magic number "startxref" at the end-of-file indicator ⓔ and reads the offset of the xref table (Step ②). It then jumps to the xref table ⓒ and checks if the table starts with the magic number "xref" (c1). The xref table is parsed so that indices of objects in the file body are stored for fast random access (Step ③). After reading the xref table, the trailer ⓓ is located and parsed as a dictionary object (Step ④). *Poppler* looks up the "root" item in the dictionary and learns that the root object of the PDF file is $\boxed{b1}$. Object $\boxed{b1}$ is fetched from the file by the index stored in the xref table. Finally, *Poppler* checks whether the retrieved object is a catalog and fetches the referred objects (according to the records in the xref table) to start rendering the PDF objects (Step ⑤). The rendering process is handled in a recursive way. For example, $\boxed{b1}$ refers to $\boxed{b2}$ and $\boxed{b3}$, which further refer to other objects, so *Poppler* renders all the referred objects recursively until no new references are found.

2.2 Limitation of existing techniques

Fuzzing. Vanilla fuzzers (e.g., AFL [3], AFL++ [18] and Mopt [31]) leverage various strategies such as genetic algorithms to randomly mutate inputs. They have found many vulnerabilities in many different applications. However, they have difficulties fuzzing PDF files. The main reason is that objects in PDFs cannot be randomly mutated as they have complex syntactic and semantic constraints. For example, a dictionary object contains multiple mandatory name-value pairs. Corrupting any pair may result in the early termination of PDF rendering. Grammar-based fuzzers (e.g., gramfuzz [2]) accept user-defined grammars and can provide syntactic guidance in mutation. However, they still have difficulties satisfying semantic constraints. For example, the length of a mutated object may be inconsistent with its content after modification, which may further corrupt all the following offsets in the xref table. Considering changing the length of $\boxed{b1}$ in Figure 1, then the offsets of the remaining 15 objects are broken. In fact, even the offset of xref table in the end-of-file indicator is corrupted. Respecting these complex semantic constraints is beyond the scope of grammar-based fuzzers. In addition, modifying $\boxed{b16}$ (an encoded stream with length 740) is also extremely challenging for grammar-based fuzzers as directly mutating the encoded raw bytes may cause failures in decoding and/or uncontrolled length changes after decoding failing the length check. To mitigate some of these problems, structure-aware fuzzers (e.g. AFLsmart [39] and Weizz [17]) utilize format specifications or learn file formats during the fuzzing process to enforce semantic constraints. However, the input formats are too complicated to be complete, and they still have difficulties dealing with the highly complex semantic constraints in PDF files. As a result, none of the existing fuzzers can automatically mutate PDF files.

Symbolic execution. Symbolic execution engines (e.g., ANGR [47] and QSYM [56]) execute programs symbolically and derive symbolic expressions for variables during execution. When branches are encountered, it forks states to explore both paths and invokes an SMT-solver to generate test cases satisfying the symbolic path constraints. In theory, it handles both syntactic and semantic constraints and can be used to generate PDF files. However, PDF files usually include lots of objects (e.g., thousands) that entail an exceptionally large number of symbolic constraints, with many falling into the hard-to-resolve categories (e.g., symbolic indexing/offsetting [30, 53]) and requiring unrolling a loop many times. Both pose hard challenges for existing symbolic execution techniques.

2.3 Our technique

We observe that for file formats with complex syntactic and semantic constraints (e.g., PDF), there are usually applications that emit such files (e.g., PDF save-as/printing). In other words, there are printer functions that convert in-memory data structures to valid files. More importantly, the aforementioned syntactic and semantic constraints are usually generated during the printing procedure, whereas data in memory are often in much-relaxed forms (e.g., fully decoded). Specifically, during parsing, the complex constraints are checked and file content is decoded into internal data structures (e.g., structs and arrays) in memory which do not have any explicit constraints (e.g., offset constraints). Instead, these constraints like those for the object offsets and the xref table offset in PDF files are constructed during printing. In the PDF reader *Poppler*, a printer function *SaveAs*() is responsible for such calculation.

Instead of directly mutating input files, we hence propose to mutate in-memory data structures and then reuse existing printer functions to emit the mutated data structures to files that are further used in testing. Specifically, our tool FuzzInMem mutates in-memory objects (e.g., delete, insert, and modify fields) for an application that processes the target file format. Then a built-in printer function of the application is reused to emit the mutated data structures to a file. The file is further used to test a *target application*. The target application may or may not be the one whose printer function is reused, called the *repurposed application*. Similar to other fuzzing techniques, coverage of the target application is used to improve in-memory mutation in the reused application. For example, the $\boxed{b1}$ in Figure 1 is parsed to a *Dict* data structure, which can be directly mutated in memory, e.g., adding an outline attribute to the dictionary. More discussion can be found in Section 3.3. Then the printer function *SaveAs*() is reused to emit the PDF. The printer function recomputes all the offsets and sizes and the mutated PDF is valid by construction. Due to its ability to produce complex inputs and perform sophisticated mutations, FuzzInMem can find deep bugs and achieve better code coverage. For example, it gets 65% more path coverage than a state-of-the-art structure-aware fuzzer Weizz and finds 20 unique bugs in PDF applications with 4 CVEs.

3 DESIGN

Figure 2 presents the overview of FuzzInMem. An application that can be repurposed for in-memory mutation usually consists of several parts, including a parser, data processor, and content printer. In part Ⓐ, FuzzInMem searches for the important in-memory structures generated during parsing files. It locates parser functions

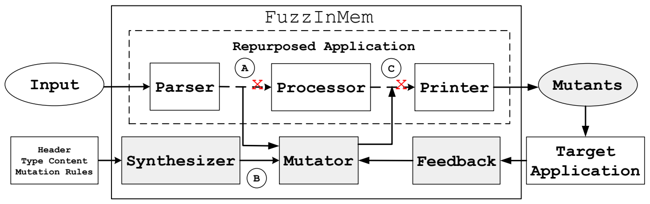


Figure 2: Overview of FuzzInMem

where in-memory structures are generated and intercepts the control flow of the application to instrument code for mutation purposes. FuzzInMem employs a synthesizer that takes source codes as well as some user-defined mutation rules like a magic number dictionary to automatically generate a mutator (Ⓑ). The mutator is instrumented into the place found in part Ⓐ, which takes the in-memory structure as input and modifies it to generate in-memory data structure mutants. The mutated in-memory structure is no longer passed to the processor but instead to the printer which FuzzInMem finds in Ⓒ by taint analysis. The printer ensures the validity (of the emitted file) as it derives appropriate sizes and offsets based on mutated structures. Note that, in many cases it can even crop mutated structures, e.g., discarding array elements that exceed the size value. The mutant is then passed to the target program and the coverage feedback is collected by the mutator to guide further fuzzing. FuzzInMem is based on AFL so it inherits the bit/byte mutations. It supports additional in-memory fuzzing by communicating with the memory mutator using pipe and shared memory.

3.1 Locate key structures

FuzzInMem needs to find the in-memory (key) structures to mutate, which represent the whole input file inside memory. Observe that most programs generate in-memory structures when parsing input files and the structures are referred to in function parameters or return values. Hence, FuzzInMem first locates parser functions in order to extract these structures. FuzzInMem finds parser functions by employing a dynamic analysis technique to get the call sequences of the application and performing a comparative analysis (of sequences) to locate the start and end of the parsing.

We develop Algorithm 1 to help identify a set of top-level user-defined functions (excluding the *main* function) that are executed during a test, in which parser functions must fall into. Comparative analysis using two different test cases is used to identify the parser function(s). Specifically, given an execution trace E of the target program, Algorithm 1 computes a call sequence ξ that only contain top level user-defined functions and a set Σ of read functions. The loop in lines 2-9 is to generate a call trace with each trace entry consisting of a function invocation and its call depth. The second loop in lines 10-16 simplifies the trace by removing library functions and user-defined functions that are not top. Whenever a call instruction is encountered, the called function is pushed into a call stack S (line 4), and the function f and its call depth $|S|$ are added to the call trace (line 5). If the function is a read API (e.g. *read* and *fread*), all the functions in the current call stack are added into the read function set Σ which contains candidate parser functions because parser functions or their parents tend to include calls to read APIs. If a return instruction is encountered, the last function is popped from the call stack (lines 8-9) to ensure validity of the

Algorithm 1: Call sequence generation.

```
Input  : execution trace of the target application ($E$)
Output : call sequence ($\xi$) and read function set ($\Sigma$)
1  $\Sigma, \xi, S \leftarrow \emptyset$
2  for $\iota \in E$ do
3      if $\iota$ == $call\ f$ then
4          $S \leftarrow S \cdot f$
5          $\xi \leftarrow \xi \cdot (f, |S|)$
6          if $f \in read$ then
7              $\Sigma \leftarrow \Sigma \cup S$
8      if $\iota$ == $ret$ then
9          $S \cdot f \leftarrow S$
10 for $(f, d) \in \xi = \xi_1 \cdot (f, d) \cdot (f', d') \cdot \xi_2$ do
11     if $f \in F_{ex}$ then
12         repeat
13             $\xi \leftarrow \xi_1 \cdot (f, d) \cdot \xi_2$
14         until $d' \leq d$;
15     else
16         $\xi \leftarrow \xi_1 \cdot (f', d') \cdot \xi_2$
17 return $\xi, \Sigma$
```

call stack. After the first loop, the call trace includes many low-level functions such as *strcmp*, *memcmp*, *malloc*, etc, FuzzInMem simplifies it by deleting these low-level functions. In particular, the functions declared in the application's header file are defined as export functions F_{ex}. FuzzInMem iterates the call sequence and deletes all non-export functions like *parseArgs*, *strcmp*, etc (line 16). FuzzInMem also removes functions whose ancestors are export functions (lines 12-14). That is, only top level export functions are retained. This step removes all sub-functions like sub parsers which parse only a specific section in the file. It is worth noting that the complex applications considered in this paper have modular designs such that their parsers are usually in parsing modules with their header files.

Next, we employ a comparative analysis to identify the start and end points of the parsing process. In our analysis, we acquire two call traces by Algorithm 1 using two inputs. Let ξ^A and ξ^B be the two call traces. The last common read function $LCRF$ is calculated according to Equation 1. In essence, it is defined as the last read function in the longest common sequence of ξ^A and ξ^B.

$$\begin{aligned} &LCS = LongestCommonSeq(\xi^A, \xi^B) \\ &LCRF(\xi^A, \xi^B) = LCS[i] \\ &\quad s.t.\ (LCS[i] \in \Sigma) \text{ and } (\forall LCS[j] \in \Sigma \Rightarrow i \geq j) \end{aligned} \tag{1}$$

We perform a two-stage process to identify the start and end of parsing. In the first stage, we find the start of the parsing function $LCRF^s$ by comparing the call sequence of execution with an invalid seed ξ^{IS} and the call sequence with a valid seed ξ^{VS}. In the second stage, we determine the end of the parsing function $LCRF^e$ by searching for the last common point of two call sequences (ξ^{VS_A} and ξ^{VS_B}) generated from two executions with different *valid seeds*. The analysis is based on the following observations: (1) Both valid and invalid seeds reach the start of the parsing stage but invalid seeds fail the parser and exit. (2) Valid seeds share the same end of the parsing stage. As a result, parser functions are the read functions between the start and end of the parsing functions in a common sequence LCS^{VV} of two valid seeds:

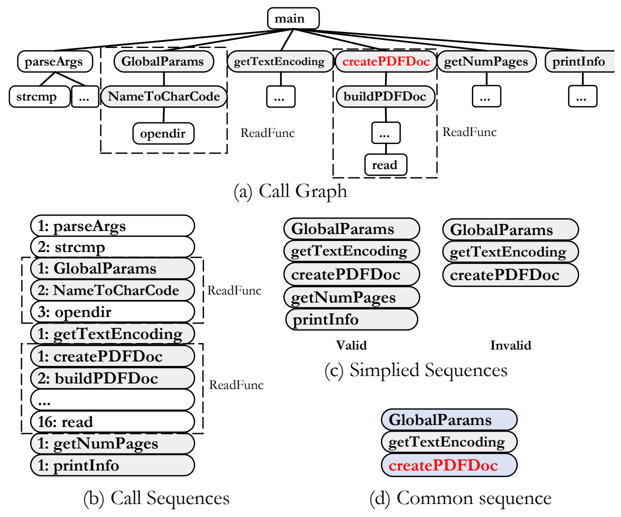


Figure 3: The process to the parser function. Grey nodes are export functions and white nodes are non-export functions.

$$
\begin{aligned}
LCS^{VV}[start] &= LCRF^{s} = LCRF(\xi^{IS}, \xi^{VS}) \\
LCS^{VV}[end] &= LCRF^{e} = LCRF(\xi^{VS_A}, \xi^{VS_B}) \\
Parser &= \{Func|Func = LCS^{VV}[i] \wedge start \leq i \leq end \wedge Func \in \Sigma\}
\end{aligned}
\tag{2}
$$

After FuzzInMem finds the parser functions, it iterates through their prototypes to get the types of parameters and the return values. The structure types we collect are called *key structures* that will be mutated later. We also find the instrumenting point for our mutator, which is right after $LCRF^e$ in the valid sequence.

Example. Figure 3 shows how we find the parser function in *Poppler*. Figure 3(a) is part of the call graph for a valid seed where the function in red is the parser function and the nodes in the dashed boxes are read functions. An execution traverses the graph in the pre-order, yielding a call sequence in Figure 3(b) with the numbers denoting the call depth. The functions in white (*parseArgs*, *strcmp* and *read*) are non-export so they are deleted. Then *NameToCharCode* and *buildPDFDoc* are deleted as their ancestors (*GlobalParam* and *createPDFDoc*) are export functions. As a result, the call sequence is simplified to the sequence in Figure 3(c). We get the longest common sequence in figure 3(d) by comparing the valid and invalid sequences. The *GlobalParams* and *createPDFDoc* are both read functions but *createPDFDoc* is the last function in the sequence so it is the start of parsing functions. In fact, *GlobalParams* does configuration and determines the encoding of the file so it is not a parser function. Note *createPDFDoc* is also the end of the parsing stage since the two valid sequences are the same in this case. Hence, we learn the only parser function is $createPDFDoc$ and its return value (whose type is *PDFDoc*∗) is the key structure.

3.2 Mutator Synthesis

In this section, we discuss how to automatically synthesize the mutator that changes in-memory structures. The idea is illustrated in Figure 4. The synthesizer takes a key data structure *Object* on the left, and derives a set of new data structures on the right, one for each structure referred to in *Object*. The new structures normalize fields in the key structure to a few *orthogonal types*, each having its corresponding mutation strategy. For example, the *GooString* field in the structure *Object* is normalized to a *string* type by removing the original typedef. As such, it is mutated by *mutate_str()* during fuzzing. Our synthesizer also generates mutation functions that cast instances of key structures to their normalized versions, traverse the normalized structures, and perform the corresponding mutations.

```
...
class Object{
    ObjType type;
    union {
        bool booln;
        int intg;
        long long int64g;
        double real;
        GooString *string;
        char *cString;
        Array *array;
        Dict *dict;
        Stream *stream;
        Ref ref;
    };
};
```

(a) Source codes

```
...
struct Ref{
    ...
};
union anony1{
    i8 booln;
    ...
    Ref ref;
};
enum ObjType{
    ...
};
class Object{
    ObjType type;
    union anony1 anonyfield1;
};
```

(b) Synthesized codes

Figure 4: Synthesize structures from source codes.

$$
\begin{array}{ll}
\langle Struct \rangle & \psi ::= \overline{(\tau, \phi)} \mid \dashv s \\
\langle Type \rangle & \tau ::= \psi \mid \tau* \mid \tau[] \mid \lambda \mid \texttt{int} \mid \texttt{float} \ldots \\
\langle Field \rangle & \phi ::= \underline{s \qquad} \langle String \rangle\ s \\
\langle Enum \rangle & \lambda ::= \overline{(s, int)}
\end{array}
$$

Figure 5: Type system for memory fuzzing.

FuzzInMem defines a basic type system to describe structures synthesized from source code as shown in Figure 5. A structure ψ is a vector of pairs, each consisting of a field τ and its type ϕ when the header file containing the structure is part of the project. That is, it is a custom type. Otherwise, if the structure is only defined in a third-party library header file, it is denoted by a structure name s. The type τ can be categorized into primitive, structure, and reference. Primitive types include four integer types, two float types, the void type, and enumeration types. An enumeration type λ is a list of strings and corresponding integer values. Reference types include arrays and pointers referring to other types. Starting from the key structures we find in 3.1, our tool synthesizes all descendent custom types that are referred to in key structures, in a recursive way. Algorithm 2 depicts how the algorithm takes a key data structure definition ψ in source code and generates the synthesized definition ψ'. Each type-field pair (τ, ϕ) in the structure definition is iterated. If a type definition is not found in the source code, we can not synthesize the field type and the compiler complains. To address this problem, we mark the field as a constant type and fill in the field hole with the exact size of the field in structure s (lines 3-4). In detail, we learn the field size from compiler debugging information and use a constant byte array to replace the original field type in the new definition. If the field is another structure type, we recursively synthesize the field type (lines 5-7). We copy the enumeration definition from the original code to make the type system complete if an enum field is found (lines 8-10). For primitive types, we simply add the type and the field into the new definition. Traversing key structures, we construct new definitions that are compatible with the original key data structure definition. This is critical as we can

Algorithm 2: Synthesize(s)

```
Input  : $\psi$: Structure definition in source code.
Output : $\psi'$: Synthesized structure definition.
$\psi' \leftarrow nil$
foreach $(\tau, \phi) \in \psi$ do
    if $\tau == \dashv s \,||\, \tau == const$ then
        $\psi' \leftarrow \psi' \cdot (\text{“}const\ byte[\mathbf{sizeof}(struct\ s)]\text{”},\ \phi)$
    else if $\tau == \psi_1$ then
        $\psi' \leftarrow \psi' \cdot (\tau, \phi)$
        $synthesize(\psi_1)$
    else if $\tau == \lambda$ then
        $\psi' \leftarrow \psi' \cdot (\tau, \phi)$
        $define(\lambda)$
    else
        $\psi' \leftarrow \psi' \cdot (\tau, \phi)$
return $\psi'$
```

cast an in-memory instance to our type. In the following, we discuss the mutation strategies for different types.

Primitive types. When mutating primitive types, FuzzInMem assigns different values to a target field. We pre-define functions for primitive types which support multiple mutation strategies, including interesting values, bit/byte flipping, shifting, adding, and random values. These operations are analogous to AFL's mutation strategies. Other mutation strategies can be easily added for the primitive types. A special case is enumeration, which is a custom type consisting of distinct values in FuzzInMem. Our synthesizer can correctly recognize enumeration types and generate special mutation functions for each enumeration type, which mutates the enumeration to every possible value.

Structure types. Structure types are essentially a reference tree, in which a child node represents a field of the structure denoted by the parent node. FuzzInMem traverses the tree and replaces the content of each node with instance values of the same type. To find candidate instances, FuzzInMem performs signature-based memory scanning. In particular, FuzzInMem instruments each composite type by adding a signature field during compilation and then scans the memory to extract all instances of each type at run time. These instances form a pool to support mutation. Containers such as vectors in C++, are special structure types. The length of a vector should always match the element number of the array. A modification to only the length or the array is likely to fail the printer. Therefore, FuzzInMem treats them as a whole and modifies the length and the array at the same time by invoking built-in APIs, which automatically observe the constraints. Nodes containing constant type qualifiers are immutable because either we do not know their internal fields or the mutation violates the semantics.

Reference types. For array types, FuzzInMem knows the exact number of elements stored in the array from the synthesizer so it can randomly select elements in the array to continue mutation. FuzzInMem also performs shuffle mutation to exchange orders of elements. The pointer of primitive types will be dereferenced and FuzzInMem mutates the underneath type without the pointer. For the pointer of composite types, FuzzInMem mutates it to point to the candidate instances of the underneath type. Note FuzzInMem also maintains a set and records the changed pointers to avoid dead loops caused by recursive reference. In addition, FuzzInMem maintains a memory pool for *int8*∗(*char*∗) so it supports magic number and string fillings.

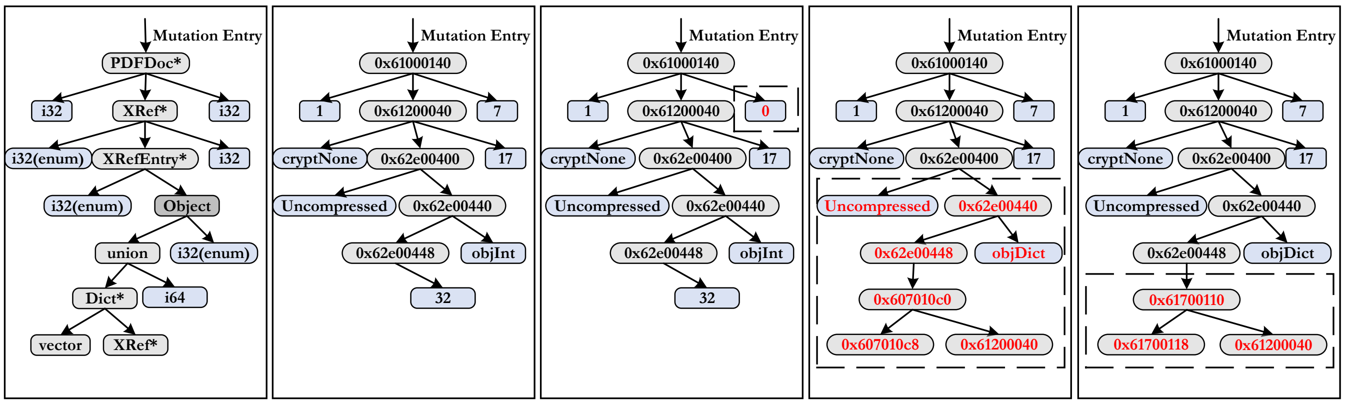
Example. Figure 6 demonstrates how a structure in PDF is mutated in memory. Figure 6(a) presents the type information of the target structure as a tree. Figure 6(b) shows an instance. We show the address if a node is a composite type and the stored values otherwise. *PDFDoc* is a structure found with the parser of *Poppler*. It has two integer fields with values 1 and 7 indicating PDF versions and a pointer pointing to the cross-reference table. The *XRef* structure includes a dynamic array *XRefEntry*∗, an integer 17 indicating the array length, and an enumeration telling the applied encryption algorithm. *XRefEntry*∗ has an enumeration describing the status of the entry and also stores its related object, which consists of a union and an enumeration telling the object type. The object in the instance is an integer with a value of 32. Starting from *PDFDoc*, we can perform primitive mutation on one of its children, changing the value from 7 to 0, which changes the PDF version from 1.7 to 1.0 (in Figure 6(c)). FuzzInMem dereferences the pointer *XRef*∗ and mutate its children. FuzzInMem samples and accesses *XRefEntry*∗ array and mutate an element. Since *XRefEntry* is a composite type, a known object collected from the memory pool is used to replace contents (in the red part of Figure 6(d)), which changes the integer object to a dictionary object. FuzzInMem can further mutates the *Dict** in Figure6(d) by reference. It replaces the whole dictionary with another instance in memory by rewriting the address of the *Dict*∗ pointer. The mutation stops when all the nodes are mutated or a maximum recursion limit is reached. Note *Dict*∗ has a field *XRef*∗ which stores the same address as the one in *PDFDoc*. FuzzInMem detects such a circle by saving the pointer values and checking at the beginning of mutating functions to avoid a dead loop. If a field mutation leads to new coverage as measured by AFL, it is considered promising and has more mutation energy.

3.3 Reuse printer function

To emit mutated in-memory data structures, FuzzInMem reuses a printer function to regenerate data files from these structures. The printer function (implicitly) checks the integrity of the in-memory structure when emitting it. If the structure is not broken, the printer outputs a new data file while following the format constraints. Otherwise, the printer fails and exits (e.g., crashes).

FuzzInMem locates printer candidates in the source code by static taint analysis using LLVM. Specifically, it looks for functions that take the key structure as the argument and outputs data that originate from the key structure. This can be achieved by marking the structure as source and the file output as sink. The analysis is lightweight, taking only a few seconds to finish.

A major challenge is that printer functions may fail in exporting in-memory structures or even crash due to failing integrity checks. We mitigate this problem by analyzing in-memory constraints. We observe that most printer failures are due to the mismatch of some invariant integer in for-loops with the length of dynamic arrays. That is when mutating an integer, the integer likely serves as the length indicator of another pointer field. Such relationships (a length integer and a pointer pointing to an array) are called dynamic arrays. Increasing the length integer alone can trigger invalid accesses to the array and crash the printer.

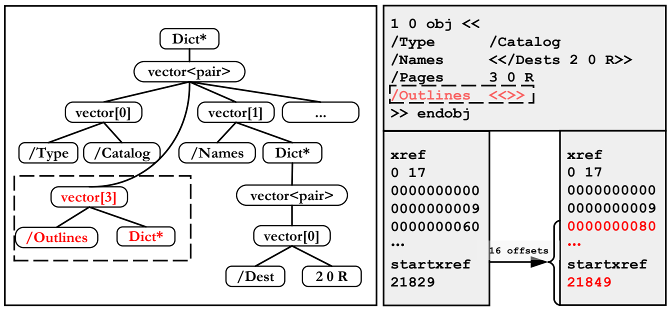


(a) Mutation Tree (b) An instance (c) Primitive mutation (d) Structure mutation (e) Reference mutation

Figure 6: Mutations of the PDF structure. Blue nodes are primitive types and grey nodes are composite types.

```
01 uxref = XRef();
02 writeHeader(outStr);
03 for(i=0; i<xref->numObjects; i++)
04   obj = xref->fetch(i);
05   offset = writeObjectHeader(i, outStr);
06   writeObject(obj, outStr);
07   writeObjectFooter(outStr);
08   uxref->add(i, offset);
09 unxrefOff = outStr->getPos();
10 writeXRefTable(uxref, outStr);
11 trailer = xref->getTrailerDict();
12 writeTrailer(trailer, outStr);
13 writeEOF(unxrefOff, outStr();
```

Figure 7: Printer function for Poppler.



```
1 0 obj <<
/Type      /Catalog
/Names     <</Dests 2 0 R>>
/Pages     3 0 R
/Outlines  <<>>
>> endobj
```

```
xref                xref
0 17                0 17
0000000000          0000000000
0000000009          0000000009
0000000060          0000000080
...                 ...
startxref           startxref
21829               21849
```

(a) Structure changes (b) File changes

Figure 8: Print a modified PDF structure.

Unlike containers, primitive dynamic arrays have no built-in APIs to help mutation, and hence we perform dynamic analysis to probe the length-array relationships. We observe that the length and the array tend to exist in the same structure near each other. For each array, we only search for the corresponding length within the same structure. We select an integer field to see if it matches the length of an array. If it does not match, the integer field is unlikely to be the length of the array. In contrast, length constraints on file inputs are usually convoluted in other arithmetic/string constraints, making them hard to satisfy.

Example. Figure 7 is the printer function we found in *Poppler* and Figure 8 shows how the printer emits a modified PDF structure to a file. During mutation, our tool changes the first object in the file and attaches a new name “*Outlines*” with an empty dictionary to the vector nodes shown in Figure 7(a). When printing the structure, the printer first emits the header, which indicates the PDF version (line 2). Then it emits the objects using a for-loop. It learns that there are 17 objects in the cross-reference table and prints the object one by one (line 3). The first object is a dictionary object so a sub-printer *writeObject()* reads the size of the vector in the dictionary and outputs each pair (line 6). Due to our constraint-preserving mutation, the length of the vector matches the number of elements so the printer can correctly print the new pair (*Outlines* in Figure 7(b)). When printing the object, the printer records the offset of each object (line 8). Then a new cross-reference is constructed by the recorded offsets (lines 9-10). The end-of-file is also recalculated and filled in at the correct place (line 13). The number in red in figure 8(b) shows the difference between the old file and the generated file. To add a new key in an object, we need to modify 16 positions to the original file to keep it valid. It is almost impossible for traditional fuzzers to perform so many mutations accurately. However, due to the valid-by-construction nature of the printer and the corresponding in-memory mutations, FuzzInMem manages all objects and offsets correctly, leading to a valid seed.

4 EVALUATION

We study and answer the following research questions:

RQ1. How popular are printer functions in applications?
RQ2. Can FuzzInMem automatically locate key structures and synthesize the codes to generate mutators?
RQ3. How does FuzzInMem compare to state-of-the-art fuzzers in terms of seed generation and code coverage?
RQ4. Can FuzzInMem help find vulnerabilities?

4.1 RQ1: Popularity of printer functions

FuzzInMem relies on the printer functions to validate and dump the in-memory modification. A straightforward question is whether printer functions widely exist in real-world applications. Observe that applications usually use open-source third-party libraries for parsing, to answer this question, we investigate a total of 71 libraries, which cover a wide range of file formats, including language parsers, image, audio/video, document, archive, font, and protocol file formats. These libraries are from Google Fuzzer's Test Suite, other fuzzing papers, or GitHub with more than 100 stars.

Our investigation results show that 67.61% (48 out of 71) libraries have printer functions. Specifically, all audio, video, and archive libraries have printer functions because they always include both encoder and decoder for the compression and depression algorithm. 90.91% document libraries and 79.16% image libraries also contain printer functions because most libraries support editing documents and images. Only a few language parsers support dumping the in-memory structures. None of the font and protocol libraries include printer functions. In conclusion, printer functions widely exist in libraries, especially in visualized file format and archive file format but rarely exist in grammar-related libraries and font libraries.

4.2 RQ2: Key structures and synthesizer results

Benchmark. Our evaluation is performed on 15 real-world applications from Google fuzzer-test-suite [4] and commonly fuzzed programs in other projects [13, 17, 18, 31, 54] for comparison purposes. We aim to cover a wide range of format categories. For each category, we select the most reputable program (i.e., having the most stars on Github). The benchmark covers various kinds of applications, including archives, images, and documents. All programs have printer functions so FuzzInMem can work on them.

Parser and key structures. We evaluate if FuzzInMem can automatically locate parsers. FuzzInMem employs *Valgrind* [33] for runtime instrumentation to obtain a simplified call sequence. A file with a randomly chosen length (30 bytes in the experiment), initialized with zeros serves as an invalid seed, and two valid seeds are used for analysis. Table 1 lists the 15 programs as well as detailed information about their parsers and key structures. 2 programs involve several parsers and printers. We only show the start of the parsers in the table and denote the parser name with * at the end. In the last column, ✓ denotes FuzzInMem can correctly find both parsers and key structures while × denotes failure. The results show that FuzzInMem can automatically locate parsers and key structures from 13 out of 15 programs. The exceptions are zlib and qpdf. FuzzInMem can correctly locate the parser function *zipOpen64* but fails in finding the key structure because the parser returns a void pointer, which does not expose information about the real return type. In this case, we find the key structure by finding the type of the return value in function definitions. For qpdf, FuzzInMem locates the wrong parser function *run* instead of *process_file* and thus the wrong key structure *QPDFJob* instead of *QPDF*. The root cause is that the program creates a wrapper class *QPDFJob* and combines the parsing and drawing functions in *run*(). However, since *QPDF* is synthesized as a field in *QPDFJob*, we can still finish the synthesis and generate mutators. In conclusion, FuzzInMem can effectively find parsers and key structures in most programs. If not, parsers and key structures are related to the found parsers, and only a few human efforts are needed to look into source codes.

Table 1: Key structures and synthesized results

Program	LOCs	Parsers	Key structure	Syn LOCs	# Types	Found
libzip	15772	zip_open	zip_t	2277	148	✓
miniz	10112	zip_open	zip_t	1313	37	✓
zlib	30878	zipOpen64	zip64_internal	1141	31	×
libpng	57175	png_read_info	png_info_def	1745	62	✓
giflib	6971	DGifOpenFileName	GIfFileType	723	14	✓
openjpeg	160791	pngtoimage	opj_image	3470	176	✓
libjpeg	88454	jpeg_read_header*	jpeg_struct	2263	81	✓
mozjpeg	91581	jpeg_read_header*	jpeg_struct	2348	83	✓
xfig	37525	read_fig	f_compound	3662	211	✓
imageMagick	382786	read	Image	9675	750	✓
mupdf	882899	fz_open_document	fz_document	18053	396	✓
xpdf	125529	PDFDoc	PDFDoc	11064	362	✓
qpdf	63168	process_file	QPDF	3386	242	×
poppler	202999	createPDFDoc	PDFDoc	11221	537	✓
podofo	63821	Load	PdfDocument	3133	174	✓

Table 2: Path coverage for 15 real-world applications

Program	AFL	AFL++	Mopt	FormatFuzz	FuzzInMem	Weizz	FuzzInMem(qemu)
libzip	249	238	244	**510**	405	192	**380**
miniz	745	665	**983**	809	944	347	**468**
zlib	349	407	401	491	**503**	205	**329**
libpng	420	475	511	N/A	**565**	474	**592**
giflib	345	266	460	**535**	501	296	**362**
openjpeg	934	N/A	938	N/A	**966**	N/A	N/A
libjpeg	2963	2717	3847	**4713**	4003	91	91
mozjpeg	2375	2359	3100	2309	**3339**	95	95
xfig	565	463	1111	N/A	**2251**	1431	**1542**
imageMagick	1833	1865	3552	**4849**	4413	N/A	N/A
mupdf	2973	2393	1171	N/A	**4109**	1644	**3876**
xpdf	2018	2407	2901	N/A	**4898**	1031	**1755**
qpdf	2565	4079	4194	N/A	**4315**	1646	**3259**
poppler	6894	6902	7339	N/A	**8771**	974	**2968**
podofo	2511	2731	2724	N/A	**3739**	742	**1403**

4.3 RQ3: Path coverage and seed generation

Competitors. We compare with 5 popular/state-of-the-art fuzzers on the benchmark. AFL [3] is the most popular fuzzing tool and serves as a baseline. AFL++ [18] is a variant of AFL that combines different scheduling and energy strategies. Mopt [31] is another popular fuzzer derived from AFL which works well in practice. For structure-aware fuzzers, we compare with Weizz [17], which automatically learns the input model from the execution trace and deduces a high-level mutation. For grammar-based fuzzers, we compare with FormatFuzzer [16], a tool that requires the user to provide an input format by composing a set of generators in C-like code. Using code to describe formats enables better expressiveness than many other grammar-based fuzzers (e.g., those that take context-free grammar specifications).

Implementation. Algorithm 1 is used to automatically find the parsers and then we manually validate the parsers and structures. The tool consists of more than 3000 lines of C code and 1500 lines of Python code to synthesize codes and generate mutation functions. Table 1 provides a breakdown of the synthesized structures and the corresponding lines of code. The generated mutation functions are integrated into each program after the parsing stage, forming a collaborative mutator with AFL to conduct fuzzing. We do not include any user-defined rules for the mutator in this evaluation.

Setup. All the fuzzers are configured with the recommended options to achieve their best performances. Since Weizz is recommended to be run in Qemu mode, we also run FuzzInMem in Qemu mode (*FuzzInMem(qemu)*) to make a fair comparison. All of our

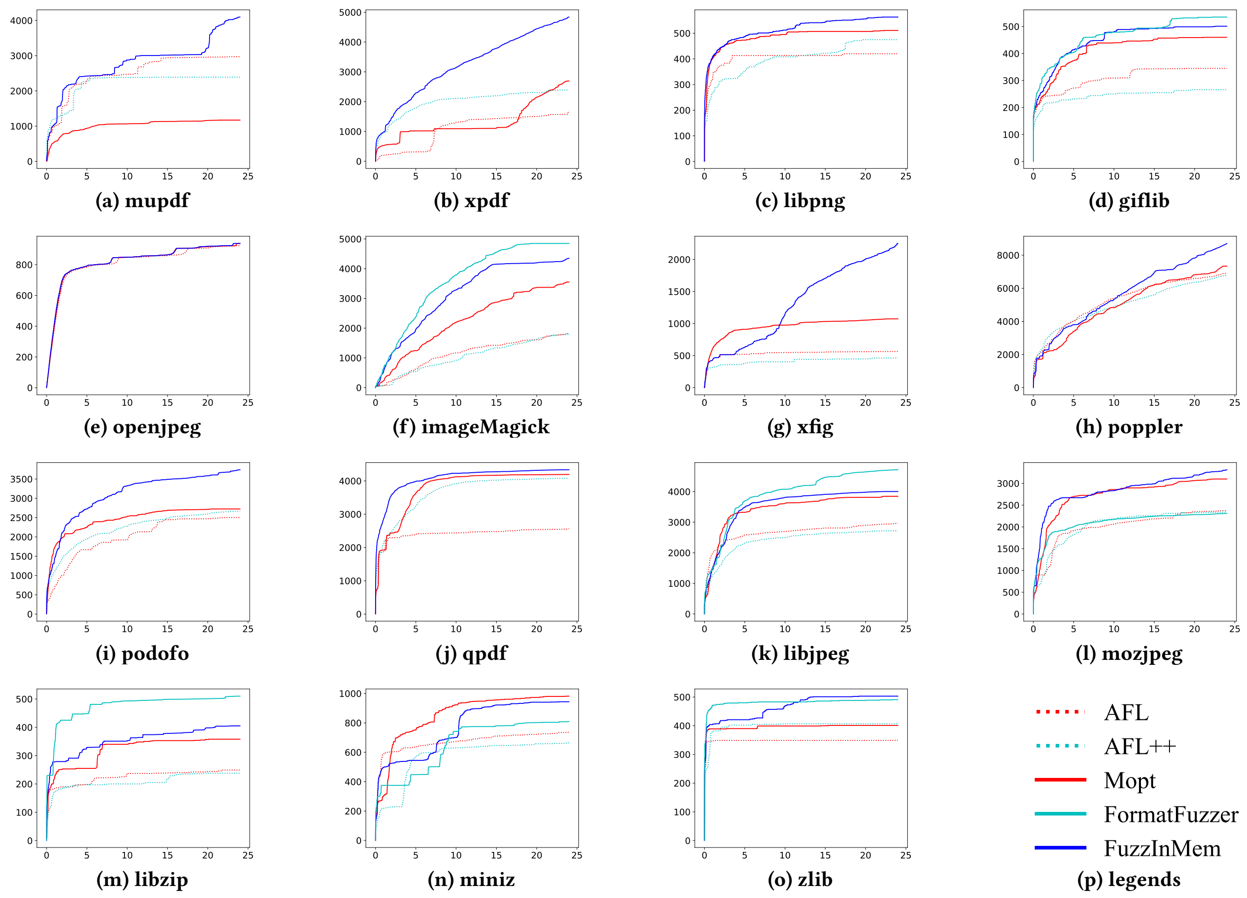


Figure 9: Path coverage. X-axis: time in hours, Y-axis: the number of unique paths.

experiments are performed on a machine with 48 cores (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz) and 196GB memory running Ubuntu 18.04 operating system. We follow the configurations recommended in a recent fuzzer evaluation work [23] and run each experiment for 24 hours with the same initial valid seeds. All the results are the median of 5 runs.

Path coverage. The results are presented in Table 2 and Figure 9. N/A in the table means the application can not be fuzzed by a tool or the path does not change due to probable under-instrumentation problems. In the figure, we do not include the curves of Weizz and FuzzInMem(qemu) because they are configured to be run in qemu mode and the coverage feedback is different. From the table and figure, we can make the following observations.

First, it is not surprising to see that AFL and AFL++ usually find the least paths. Although adopting different seed scheduling strategies, their curves look similar in most cases, which means the seed scheduling strategies can not make a difference in 24 hours. Second, Mopt works well for 11 out of 15 programs so the path coverage outperforms AFL and AFL++. However, it still suffers from the problem that the fuzzer is stuck in parser errors and finds fewer paths than FuzzInMem in most cases. The reason why Mopt outperforms AFL and AFL++ is that it calculates the probability for each mutation method and chooses the ones more likely to find new paths. Third, Weizz gets good path coverage for chunk-based applications because it quickly learns the format from the execution traces. Nonetheless, the heuristic rule does not work well for document applications so it fails to do structure mutation. In addition, the analysis to learn the input model is rather expensive which makes the fuzzing inefficient and leads to poor path coverage.

While FormatFuzzer demonstrates its efficacy in fuzzing recognized formats and surpasses FuzzInMem in 4 programs, its applicability is limited to only 7 out of 15 programs due to its support for a restricted set of formats. Notably, FormatFuzzer lacks support for the PDF format, a widely recognized and utilized document format. Although its performance is comparable to FuzzInMem in the formats it supports, FormatFuzzer requires the user to have extensive domain knowledge in order to provide format generators. Finally, FuzzInMem achieves the best path coverage in 10 programs out of 15. On average, FuzzInMem outperforms AFL, AFL++, Mopt, Weizz and FormatFuzzer by 70.45%, 66.64%, 41.86%, 65.08% and 1.82%, respectively. FuzzInMem works particularly well for several programs (e.g. mupdf, podofo, and xfig) because it successfully generates different interesting seeds from the initial seed sets. For example, FuzzInMem generates a new pdf file that separates the cross-reference table into several parts by modifying an enumeration and thus contributes to different parsing logic.

Table 3: Percentage of valid seed generated by fuzzers

Program	AFL	AFL++	Mopt	Weizz	FuzzInMem
libzip	5.22e-4	3.30e-4	4.18e-5	7.25e-4	**6.80e-1**
miniz	3.27e-4	9.46e-4	5.01e-4	1.48e-3	**6.39e-1**
zlib	3.75e-2	4.02e-2	8.33e-2	1.11e-1	**7.92e-1**
libpng	1.25e-6	1.44e-5	1.26e-6	6.93e-4	**6.93e-1**
giflib	9.83e-3	3.66e-2	4.77e-3	3.87e-4	**3.81e-1**
openjpeg	4.63e-1	N/A	3.11e-1	N/A	**7.67e-1**
libjpeg	3.49e-5	6.44e-4	1.31e-4	2.82e-5	**8.52e-1**
mozjpeg	5.38e-5	6.09e-4	1.41e-4	2.83e-4	**8.09e-1**
xfig	7.52e-2	1.17e-2	8.60e-2	8.51e-3	**7.53e-1**
imageMagick	6.24e-2	8.50e-2	1.96e-4	N/A	**8.36e-1**
mupdf	1.23e-1	1.54e-1	7.51e-2	8.17e-1	**8.94e-1**
xpdf	1.54e-1	2.33e-1	1.17e-1	7.34e-1	**8.32e-1**
qpdf	2.31e-4	5.29e-4	3.50e-4	1.43e-1	**8.16e-1**
poppler	2.04e-1	2.96e-1	1.07e-1	2.45e-2	**7.57e-1**
podofo	1.91e-2	3.30e-2	1.56e-2	8.26e-2	**7.23e-1**

Seed generation. We further study the seed generation capability of FuzzInMem against other fuzzers. We measure the percentage of generated files that remain valid throughout the mutation. We configure FuzzInMem to perform only in-memory mutation and disable plain bit/byte mutations. Table 3 shows the result. Data for FormatFuzzer is omitted as it inherently knows formats and consistently generates valid seeds for known formats. N/A means a tool fails to fuzz the application. We employ the corresponding fuzzed application to measure if the seed is accepted and processed correctly. In this experiment, a seed is considered valid if the target application returns 0.

We can learn that all fuzzers can generate valid seed inputs during fuzzing. Different seed scheduling strategies (AFL vs. AFL++) can affect the number of generated valid seeds. In most applications, AFL++ outperforms AFL in seed generation. The only exceptions are libzip and xfig. Mutation chosen strategy (AFL vs. Mopt) also impacts valid seed generation. Mopt generates much fewer seeds in libzip and imageMagick because it keeps applying the same mutation and focuses on exploring the error-handling codes in parsers. Thanks to the structure-aware mutations, Weizz generates hugely more valid seeds than mutation-based fuzzers in some applications like qpdf and xpdf. It also generates some valid seeds for libpng while other fuzzers can hardly generate any because it learns the chunk structures in png file format on the fly. However, it can not pass difficult checks such as crc32 check so the percentage of valid seeds is still low in libpng. In contrast, FuzzInMem can generate the most percentages of valid seeds in all applications because the printers are constraint-preserving, which automatically follows format constraints. In most applications, FuzzInMem outperforms its competitors by 10x-10000x times. However, even FuzzInMem can not guarantee to generate valid seeds because the parsers and processors in applications can reason about the semantic meanings of file inputs. For example, if FuzzInMem changes a dictionary object to an array object, though the printer can dump the in-memory structure, the file can still be rejected because the processor accepts only dictionary or integer objects in a predicate. In conclusion, FuzzInMem can outperform other fuzzers in most programs for both path coverage and valid seed generation.

4.4 RQ4: Bug detection

To measure the bug-finding capabilities, we keep FuzzInMem running for a week on the latest version of each tested program. FuzzInMem can successfully detect 29 unique vulnerabilities that are previously not reported. We are also awarded 5 CVEs for exploitable bugs. All 29 bugs are confirmed by developers and 25 of them are already fixed. The vulnerabilities cover a wide range of categories, including stack exhaustion, use after free, heap overflow, stack overflow, null pointer dereference, and memory leak. FuzzInMem can identify unique bugs due to its ability to explore more program paths by changing in-memory structures. We show a vulnerability found by FuzzInMem as a case study.

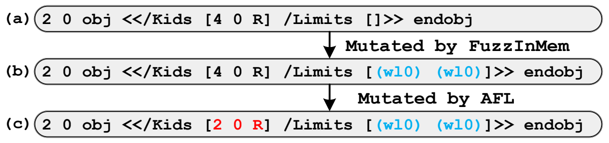
Case study I. This vulnerability is found in *Xfig* , which is an open-source vector graphics editor. It supports different kinds of shapes, including arcs, ellipses, polylines, splines, etc. Each shape can have sub-types. For example, a polyline can be a `box`, `polygon`, `arc box`, or `imported-picture bounding box`, which is denoted as an enumeration in memory. The polygon includes at least 5 points and a box includes only 4 points. When mutating a data structure *F_line*, it changes the subtype of a polyline from a `box` to a `polygon`. The printer adaptively converts the shape to a polygon with four points. The generated file can be still accepted by the parser. However, when *Xfig* tries to process it and converts it to a GBX file format (Gerber format), a null pointer dereference is triggered. The root cause is that *Xfig* assumes a polygon has at least 5 points and accesses the fifth point without any validation checks. Other fuzzers are not likely to find the bug because they can not easily change a shape from box to polygon by bit/byte mutations.

Case study II. Figure 10 presents a vulnerability in *Xpdf* , which exists for more than 10 years and affects many commercial applications such as Apple Preview, Foxit PDF Reader, and Adobe PDF Reader. The function $findDestInTree()$ tries to search for a name in the tree dictionary and retrieve the corresponding destination object, which is further used for page jumping. The function first looks for "Names" array in the tree dictionary (Line 5) and compares the "name" argument to elements in "Names" array (Lines 6-8). If it finds a match, it returns the corresponding destination object (Line 9). Otherwise, the function tries to find "name" in the "Kids" array of the tree (Lines 10-12). It iterates the "Limits" array in each kid and figures out the range of names in the kid. If "name" lies in the limit range, it searches the "name" in the kid tree by the recursive call (Lines 13-18). If nothing is found, the function returns a null object indicating no object found (Line 19). However, Line 18 is vulnerable and exposes a stack exhaustion bug.

The input file in our motivation example (figure 1) triggers the bug by infinite recursion. The vulnerable function tries to find the name "wl0" in the tree. But the tree does not contain "Names", hence it should find the name "wl0" in "Kids". It accesses kids and finds "wl0" is in the limit range (from "wl0" to "wl0"), so the function makes a recursive call to find "wl0". However, the kid of the object is itself, forming a self-loop that leads to infinite recursion and finally crashes the program. The bug can be uniquely found by the cooperation of FuzzInMem and AFL. Figure 11 shows parts of the mutation history. Originally, the input shown in Figure 11(a) contains an object whose "Limits" is an empty array. When mutating the in-memory structure, FuzzInMem adds two "wl0" to the empty

```
Object findDestInTree(
       Object *tree,
       GString *name,
       Object *obj){
if(tree->dictLookup("Names", &names)->isArray())
  for(i = 0; i < names.arrayGeyLength(); i+= 2)
    if(names.arrayGet(i, &name1)->isString()
    && name->cmp(name1.getString() == 0))
      return obj = names.arrayGet(i+1, obj);
if (tree->dictLookup("Kids", &kids)->isArray())
  for (i = 0; i < kids.arrayGetLength();i++)
    if(kids.arrayGet(i, &kid)->isDict())
    if(kid.dictLookup("Limits", &limits)->isArray())
    if(limits.arrayGet(0, &low)->isString()
    && name->cmp(low.getString() >= 0
    && limits.arrayGet(1, &high)->isString()
    && name->cmp(high.getString() <= 0)
      findDestInTree(&kid, name, obj);
return obj->initNull();
}
```

Figure 10: A vulnerability in Xpdf.


(a) `2 0 obj <</Kids [4 0 R] /Limits []>> endobj`

(b) `2 0 obj <</Kids [4 0 R] /Limits [(wl0) (wl0)]>> endobj`

(c) `2 0 obj <</Kids [2 0 R] /Limits [(wl0) (wl0)]>> endobj`

Figure 11: Cooperation of FuzzInMem and AFL to expose bug.

array and modifies the corresponding array length, generating the mutants shown in Figure 11(b). Since the mutants lead to new path coverage, it is further mutated by AFL, which changes "`4 0 R`" to "`2 0 R`" by bit/byte mutations, generating the mutant shown in Figure 11(c). Eventually, all the pre-conditions are satisfied and the bug is triggered.

5 LIMITATIONS

FuzzInMem cannot fuzz programs that do not have any initial valid inputs. FuzzInMem relies on parser functions to transform inputs into in-memory structures. In the absence of initial valid inputs, the parser fails, preventing our analysis (e.g., identifying in-memory structures). Besides, FuzzInMem needs printer functions thereby restricting its applicability to programs without such functions. Fortunately, we observe that many popular We only need one good printer for each format, as the printer may be chained up with other consumer applications of the same format to form an end-to-end fuzzing pipeline.

6 RELATED WORK

Fuzzing. Fuzzing is a promising technique for vulnerability discovery. It can be categorized into generation-based [1, 6] and mutation-based [3, 5]. There exist a large number of studies aim to improve the effectiveness of fuzzing by learning the input format offline [20, 49] or on-the-fly [9, 30, 40, 53, 54], and dynamically adapt generation/mutation strategies based on the fuzzing history [11, 18, 26–29, 31, 37, 41, 57].

Closely related works are those that mutate objects other than input files. Parametric fuzzing, as introduced in previous works such as [34–36], represents a pioneering technique for conducting indirect mutations. For example, Zest [35] leverages a generator function that takes a sequence of bits as a parameter to generate valid output files. It mutates the parameter so that a small change of the parameter is projected to significant changes in output files. The difference is that parametric fuzzing requires a well-crafted generator function but FuzzInMem reuses built-in printers. We consider that FuzzInMem and parametric fuzzing are complementary. Specifically, in scenarios involving complicated formats like PDF, where crafting an effective generator may be impractical for non-experts, FuzzInMem offers a more accessible alternative. In addition, parametric fuzzing could be easily used to generate a certain part of an input, e.g., an image embedded inside a PDF file. Some tree-based grammar fuzzers/generators [38, 50] use provided grammars to parse test cases into Abstract Syntax Trees and then mutate them at the subtree level. Although FuzzInMem also performs tree-based mutation, our trees do not reflect syntactic structures but rather memory object references. To mitigate the dependency on predefined grammars, researchers have suggested enhancing fuzzers/generators by employing machine learning techniques to learn formats from inputs [20, 43]. Learn & Fuzz [20] applies a sequence-to-sequence technique to learn the grammar of objects from an extensive PDF file corpus. As a result, it can generate correct objects. Subsequently, with a manually crafted printer function, these objects are arranged and printed to form a syntactically correct PDF file. FuzzInMem, on the other hand, does not need a large corpus or a manually crafted printer to fuzz PDF files. Another line of work is to fuzz generators. Fuzztruction [8] and MutaGen [22] mutate existing format generators to emit semi-valid seeds. FuzzInMem differs from these techniques by mutating the in-memory structures instead of the generators themselves.

Application Logic Reuse. At the core of FuzzInMem is application logic reuse, which makes certain program analysis tasks easier to achieve. BCR [12] extracts malware encryption and decryption functions and reuses them in other programs. Inspector Gadget [24] automatically identifies the instructions that are responsible for specific malware behavior for further reuse and analysis. Virtuoso [15], VMST [19] and Hybrid-Bridge [44] identify logic (from in-guest applications or the whole system) that could be reused for virtual machine introspection. DSCRETE [46] and RetroScope [45] leverages the existing code for memory content rendering of a single data structure or full display screens. FuzzInMem transfers the philosophy of application logic reuse to program testing. In particular, FuzzInMem leverages the built-in printer functions to fix the constraints broken by mutations.

7 CONCLUSION

We observed that well-designed applications have built-in print functions that preserve constraints and can be used to generate valid input files. Based on this observation, we developed FuzzInMem, a novel technique that leverages the logic of these print functions. FuzzInMem includes a key structure analyzer, a mutation synthesizer, and a printer. It identifies key structures and the appropriate location for instrumentation through differential analysis of call sequences. The source code is then modified to include a mutator and perform tree-based mutations in memory. Finally, the print function is utilized to verify the integrity and generate the structures as files. Our evaluation demonstrates the effectiveness of FuzzInMem in terms of path coverage, seed generation, and bug detection.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments. The Purdue authors were supported, in part by DARPA VSPELLS - HR001120S0058, NSF1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. The RUC authors were supported, in part by the National Natural Science Foundation of China (NSFC) under grant 62002361, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors

REFERENCES

[1] 2010. SPIKE Fuzzer. http://resources.infosecinstitute.com/intro-to-fuzzing.
[2] 2020. GramFuzz. https://github.com/d0c-s4vage/gramfuzz.
[3] 2021. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl.
[4] 2021. Google Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.
[5] 2021. libfuzzer. https://llvm.org/docs/LibFuzzer.html.
[6] 2023. Peach Fuzzer. https://www.peach.tech/products/peach-fuzzer.
[7] 2023. Poppler. https://poppler.freedesktop.org.
[8] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. 2023. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *32st USENIX Security Symposium (USENIX Security 23)*. USENIX Association.
[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 1985–2002.
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*.
[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security CCS 2016*.
[12] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. 2010. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society. https://www.ndss-symposium.org/ndss2010/binary-code-extraction-and-interface-identification-security-applications
[13] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP 2018)*.
[14] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade Farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *International Symposium on Recent Advances in Intrusion Detection*.
[15] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon T. Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 297–312. https://doi.org/10.1109/SP.2011.11
[16] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2021. FormatFuzzer: Effective Fuzzing of Binary File Formats. *CoRR* abs/2109.11277 (2021). arXiv:2109.11277 https://arxiv.org/abs/2109.11277
[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3395363.3397372
[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.
[19] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 586–600. https://doi.org/10.1109/SP.2012.40
[20] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. https://doi.org/10.1109/ASE.2017.8115618
[21] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795
[22] Ulf Kargén and Nahid Shahmehri. 2015. Turning Programs against Each Other: High Coverage Fuzz-Testing Using Binary-Code Mutation and Dynamic Slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 782–792. https://doi.org/10.1145/2786805.2786844
[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804
[24] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. 2010. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 29–44. https://doi.org/10.1109/SP.2010.10
[25] Gwangmu Lee, Woo-Jae Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *USENIX Security Symposium*.
[26] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874
[27] Caroline Lemieux and Koushik Sen. 2017. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. *CoRR* abs/1709.07101 (2017).
[28] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176
[29] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295
[30] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: Facilitating Seed Input Generation in Fuzzing via String Constraint Solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 391–403. https://doi.org/10.1145/3533767.3534403
[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu
[32] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. 2022. SLIME: Program-Sensitive Energy Allocation for Fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 365–377. https://doi.org/10.1145/3533767.3534385
[33] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746
[34] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002
[35] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576
[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Validity Fuzzing and Parametric Generators for Effective Random Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 266–267. https://doi.org/10.1109/ICSE-Companion.2019.00107
[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints.

Proc. ACM Program. Lang. 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

[38] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[39] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[40] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR* abs/1711.04596 (2017).

[41] Sanjay Rawat, Vivek Jain, Ashish KumVuzzerar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017.*

[42] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 861–875.

[43] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399

[44] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-Bridge: Efficiently Bridging the Semantic-Gap in VMI via Decoupled Execution and Training Memoization. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.* The Internet Society. https://www.ndss-symposium.org/ndss2014/hybrid-bridge-efficiently-bridging-semantic-gap-virtual-machine-introspection-decoupled

[45] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G. Richard III. 2016. Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 1137–1151. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/saltaformaggio

[46] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2014. DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 255–269. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/saltaformaggio

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium, NDSS 2016.*

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[50] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[51] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. *Proceedings 2021 Network and Distributed System Security Symposium* (2021).

[52] Wikipedia. 2021. PDF. https://en.wikipedia.org/wiki/PDF.

[53] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080

[54] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*. 769–786. https://doi.org/10.1109/SP.2019.00057

[55] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 130, 18 pages.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018*. 745–761.

[57] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. 659–676. https://doi.org/10.1109/SP40001.2021.00109