ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery

Wei You[1], Xueqiang Wang[2], Shiqing Ma[1], Jianjun Huang[3], Xiangyu Zhang[1], XiaoFeng Wang[2], Bin Liang[3]
[1]Department of Computer Science, Purdue University, Indiana, USA
[2]School of Informatics, Computing, and Engineering, Indiana University Bloomington, Indiana, USA
[3]School of Information, Renmin University of China, Beijing, China
Email: {you58, ma229, xyzhang}@purdue.edu, {xw48, xw7}@indiana.edu, {hjj, liangb}@ruc.edu.cn

***Abstract*—Existing mutation based fuzzers tend to randomly mutate the input of a program without understanding its underlying syntax and semantics. In this paper, we propose a novel on-the-fly probing technique (called ProFuzzer) that automatically recovers and understands input fields of critical importance to vulnerability discovery during a fuzzing process and intelligently adapts the mutation strategy to enhance the chance of hitting zero-day targets. Since such probing is transparently piggybacked to the regular fuzzing, no prior knowledge of the input specification is needed. During fuzzing, individual bytes are first mutated and their fuzzing results are automatically analyzed to link those related together and identify the type for the field connecting them; these bytes are further mutated together following type-specific strategies, which substantially prunes the search space. We define the probe types generally across all applications, thereby making our technique application agnostic. Our experiments on standard benchmarks and real-world applications show that ProFuzzer substantially outperforms AFL and its optimized version AFLFast, as well as other state-of-art fuzzers including VUzzer, Driller and QSYM. Within two months, it exposed 42 zero-days in 10 intensively tested programs, generating 30 CVEs.**

I. Introduction

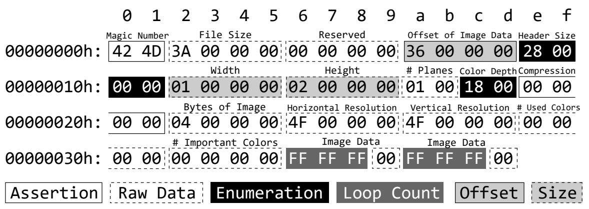
Fuzzing is a widely-used software testing technique that runs randomly or strategically generated inputs against a program to find its vulnerabilities. It has long been considered to be one of most effective ways in program security analysis, contributing to the discovery of most high-impact security flaws, e.g., CVE-2014-0160, CVE-2017-2801 and CVE-2017-3732 [1], [2]. Critical for such a dynamic analysis is the production of inputs with a high degree of code coverage and good chances of hitting the security weaknesses hiding inside the code. Typically this has been done either through *generating* legitimate inputs using the knowledge of an input model, or by *mutating* a set of initial seed inputs, for example, through randomly flipping their bits or strategically substituting known values for the input content. These conventional approaches, however, become increasingly inadequate in the presence of the modern software characterized by a large input space and complicated input-execution relations.

Challenges. More specifically, the generation-based approach relies on a grammar describing the input format to produce legitimate test cases. The grammar here needs to be given a priori [3], [4] or recovered from execution traces off-line [5], [6]. More importantly, although a well-constructed grammar indeed reduces the search space, it offers little clue about the impacts that various inputs can have on the program's execution: for example, whether a set of inputs will all lead to the same execution path and therefore only one of them should be tested. Furthermore, inputs that exploit vulnerabilities may not even follow the input grammar because an implementation may choose to ignore certain input fields if they are irrelevant to the functionalities (e.g., an image format converter may ignore fields related to rendering); also buggy implementations may even accept ill-formed inputs.

The mutation-based approach does not require the knowledge about input model. Instead, it utilizes a set of legitimate input instances as seeds and continuously modifies them to explore various execution paths. Prior research has studied the way to choose the right seeds that likely lead executions to vulnerable program locations [7], [8], and the way to determine critical input bytes that likely enhance code coverage [9], [10]. For a given seed, however, mutation is typically random, which does not scale even when the input data size is moderate.

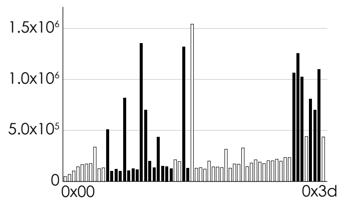
As we can see here, essential to a highly effective and scalable fuzzing process is in-depth understanding of the target program's inputs. Such an input is often structured data and can be partitioned into a sequence of data fields with specific semantics: for example, buffer size, category indicator, etc. The semantics of these fields are invaluable for vulnerability discovery, helping a fuzzer determine the fields to mutate, the range of legitimate values, the impact of a data field on program execution, and so on. Leveraging such information, the fuzzer can operate in a more intelligent way, focusing only on the subspace of inputs most likely leading to execution paths never seen before. This information (fields and their semantics), however, may not be documented and available to the fuzzer, and can be hard to recover without going through an in-depth heavyweight analysis procedure.

Fuzzing with Type Probing. To address the challenges and elevate today's fuzzing technique using rich input semantics, we present a new smart fuzzing technique, called *ProFuzzer*, which automatically discovers input fields and their semantics through a lightweight random fuzzing procedure called *probing*, to guide online evolution of seed mutations. Such probing is entirely piggybacked to regular fuzzing and executed on the fly. More specifically, ProFuzzer performs a two-stage fuzzing on a target program through mutating a set of seeds.

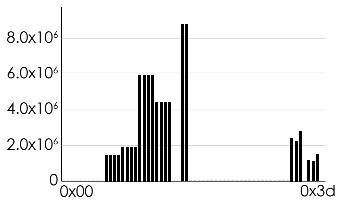


```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 3A 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010h: 00 00 01 00 00 00 02 00 00 00 01 00 18 00 00 00
00000020h: 00 00 04 00 00 00 4F 00 00 00 4F 00 00 00 00 00
00000030h: 00 00 00 00 00 00 FF FF FF 00 FF FF FF 00
```

(a) Sample bmp image.



(b) Mutations (AFL).



(c) Mutations (ProFuzzer).

Fig. 1: Motivation example.

During the first stage (i.e., the probing stage), it conducts sequential byte-by-byte probing, that is, changing one byte each time, enumerating its values against the target program and then moving onto the next byte. This sequential fuzzing step also collects information about the target's execution paths under different byte values. The information is automatically analyzed online to recover data fields (linking related bytes together) and determine their *field type*: for example, the consecutive bytes whose specific content leads to the same exception path under mutation can be grouped into a field. In our research, we identified 6 field types that model the way their content affects the program's execution, such as *size*, *loop count*, and *enumeration* that has only a few valid values correctly interpreting its subsequent content on the input. These types are application agnostic, describing fuzzing related semantics. The discovery of data fields and their types offers guidance to fuzzing at the second stage, during which ProFuzzer mutates each field to *exploit* the values that could lead to an attack (e.g., a large data size that may exploit a buffer-overflow vulnerability), and *explore* legitimate values according to the field type for better coverage.

We implement the design on AFL [11]. We compare ProFuzzer with AFL, AFLFast [7], a state-of-the-art program analysis aided fuzzer VUzzer [12] and two state-of-the-art hybrid fuzzers Driller [13] and QSYM [14] on 4 real applications with 20 known vulnerabilities and two standard benchmarks (LAVA-M [15] and the Google fuzzer test-suite [16]). The results show that ProFuzzer is able to discover more bugs with less time. For example, for the 4 real applications, ProFuzzer can discover 18 bugs in 24 hours while the best result of other fuzzers is only 15. In addition, the average time needed by ProFuzzer for each bug is only 18-73% of the time needed by other fuzzers. More details can be found in Section V.

Discoveries. We further ran ProFuzzer for two months on 10 popular real-world open-source applications, including the software for image processing (e.g. OpenJPEG), audio and video decoder (e.g., LibAv), PDF reader (e.g., MuPDF) and compression library (e.g., ZzipLib). Our study has resulted in the discovery of 42 zero-day vulnerabilities, such as heap overflow, infinite loop, etc. Most of them will have serious consequences once exploited. For example, a heap buffer over-read vulnerability in Exiv2 (CVE-2017-17725) can be triggered through a very small crafted image file, leading to 4 bytes information leak that facilitates further exploit (e.g., bypassing address space randomization). Note that even though Exiv2 has been extensively tested [17], this vulnerability still fell through the crack. We have reported our findings to all the affected software vendors. So far, most of them have acknowledged that the problems we found are real and significant. We have been awarded 30 CVEs and the investigations on other reported problems are ongoing.

Contributions. We have the following contributions:

• *New smart fuzzing technique*. We designed an efficient smart fuzzer, which discovers the relations between input bytes and program behaviors via lightweight probing, to guide follow-up targeted seed mutations. This approach seamlessly and transparently integrates semantic knowledge discovery into a fuzzing process, improving not only accuracy of the fuzzing but also its performance, without degrading applicability.

• *Discovery of zero-day vulnerabilities*. Running our fuzzer on real-world applications, we discovered 42 zero-day vulnerabilities in two months. Our findings demonstrate that this online semantic discovery and mutation guiding technique improve the state-of-the-art.

II. Motivation

We use OpenJPEG [18] to explain the problems of traditional fuzzing techniques and the idea of ProFuzzer. OpenJPEG is an open-source image codec library that converts various image formats into JPEG 2000. It is widely integrated into commodity software for image processing.

Mutation-based Fuzzing. Mutation-based fuzzing mutates seed inputs without knowing their semantics. As such, mutations are largely applied randomly. The approach is simple and application agnostic. However, it does not work effectively due to the lack of input semantics. Figure 1a shows a seed input for OpenJPEG. We use AFL, the most widely used mutation-based fuzzing tool, to fuzz an old version OpenJPEG library with 5 known vulnerabilities and after 24 hours, only 3 of them are discovered. We found that only 24 bytes out of 62 (highlighted in Figure 1a) are helpful for coverage improvement. Figure 1b shows the number of mutations on each byte within 24 hours. More than 60% of the mutations are not helpful. The root cause is that in the absence of input specifications, mutation-based fuzzing does not receive proper guidance. There are existing efforts to infer input semantics for improving fuzzing. AFL provides a simple off-line file format analyzer utility (AFL-analyze [19]) to infer the input syntax and a built-in dictionary construction functionality [20] to identify application-specific keywords. However, they have limited effectiveness (see Section V-C) due to their simplicity.

Specification-based Fuzzing. A natural enhancement of the AFL-like approach is to use input specifications (e.g., input syntax and semantic constraints) to guide fuzzing. Indeed a prior study on symbolic execution based testing has shown that input grammar can help substantially improve the effectiveness of fuzzing [21]. However, these approaches are often difficult to deploy. First, specifications are not always available. Second, specification can be implicit and requires a lot of human efforts to extract. This greatly affects the applicability of the fuzzing technique. Most importantly, fuzzing is a per-binary process. It is highly related with real implementation, which may have nontrivial deviation from their input specification. Applications usually only support part of the specification, and sometimes specification will include some reserved bytes whose functionalities are customized in various implementations/extensions [22]. For example, the bytes from offset $0x1c - 0x1d$ in Figure 1a are used to represent color depth. Their possible values, according to the specification [23], are 0x01, 0x04, 0x08, 0x10, 0x18 and 0x20. However, the current version of OpenJPEG does not support 0x01 or 0x04. Finally, these specifications are not fuzzing-oriented, thus many of the generated inputs are still redundant for fuzzing. Namely these inputs are valid for the program, but cannot improve path coverage or vulnerability discovery.

Basic Idea of ProFuzzer. ProFuzzer is inspired by two important observations. First, *a comprehensive, application-specific and semantically rich input specification is not necessary for fuzzing.* For instance, the bytes from offset $0x26$ to $0x2d$ in Figure 1a indicate the image's resolution. They are important input fields according to the file format specification. However, OpenJPEG skips these bytes during image conversion. Therefore, identifying the field does not benefit fuzzing much as their content does not affect the execution path during conversion. Instead, the more useful information is to recognize fields of special types that are critical for fuzzing. For instance, if we know that a field describes a buffer size, we can apply customized mutation rules to find out whether a vulnerable buffer is present. Second, *inputs can be understood and their fields and data types can be discovered by directly observing the fuzzing process, particularly the ways the input content is mutated and the program's execution path variations in response to the mutations.* As such, existing input format reverse engineering techniques (e.g., REWARDS [24], TIE [25], Howard [26]) that rely on heavy-weight analysis are neither necessary nor cost-effective. Instead, we believe that input understanding can be seamlessly integrated into the fuzzing process, through on-the-fly semantic analysis on prior fuzzing results to recognize input field and their types, and then guide future input mutations.

Following the observations, we design a two-stage fuzzing technique, ProFuzzer. During the first stage, it probes the input fields and field types of the seed input through per-byte mutations. Particularly, based on the behavior variations of the target program (e.g., execution path changes) under mutations, ProFuzzer can recognize input fields that are critical. The field and field type information is then used in the second stage to guide online input mutations. Different field types have different priorities and entail various mutation policies. For example, our approach avoids mutating fields of the *raw data* type, whose mutations cause the target program to go through the same path. For fields with the *buffer size* type, the multiple bytes belonging to the same field are mutated together, following specific mutation patterns such as prioritizing boundary values. In contrast, even though AFL has some pre-defined mutation patterns such as changing a random subsequence of consecutive input bytes to special values (e.g., 0xFF), it can hardly apply them to the right subsequences.

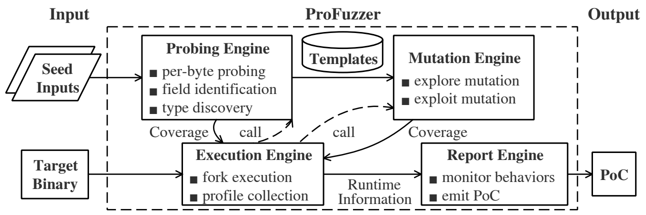


Fig. 2: ProFuzzer architecture.

With field-aware mutations, ProFuzzer improves both effectiveness and performance of fuzzing. We run ProFuzzer for 24 hours on OpenJPEG, discover all 5 known vulnerabilities and achieve a path coverage 60% more than AFL. We also calculate the number of mutations that each byte goes through during the 24 hours in Figure 1c. Note that consecutive bytes in the same fields have the same mutation count. Specifically, only 24 out of 62 input bytes have been mutated, since they are in the fields important to fuzzing. For example, ProFuzzer probes and determines that an *offset* field is present at $0x0e$ to $0x11$, which determines the offset of the subsequent data bytes (from the file header). Offset fields are critical for vulnerability detection due to their substantial impact on memory behaviors. This information allows ProFuzzer to avoid blind fuzzing and instead use a specific mutation policy (e.g., setting the value of an offset field to be the difference between the current location and the end of file) to strategically explore the input space of the field. For another example, the bytes at $0x1a$ and $0x1b$ are recognized as raw data after 512 probing runs and therefore there are not further mutations on those bytes, whereas they are mutated 407,170 times by AFL.

III. Overview

The architecture of ProFuzzer is shown in Figure 2. It consists of four components: the probing engine, the mutation engine, the execution engine and the report engine.

Probing Engine. The task of the probing engine is to generate type templates from seed inputs for a given binary. For each seed input, ProFuzzer goes through all the bytes. For each byte, it iterates through all possible values ($0x00$ to $0xff$) and collects the corresponding execution profiles. It then extracts some metrics (e.g., edge coverage variation caused by mutation) from these profiles, and uses them as the features of the byte. Then, ProFuzzer groups the consecutive bytes with similar features into a field. Finally, it determines the type of each field based on the relations between the changes of its possible values and the corresponding variations of observed features (e.g., a magic number field can be characterized by the pattern that all mutations happening to the original value in the seed input lead to invalid inputs and hence short executions).

Mutation Engine. The mutation engine uses the probed template (i.e., fields and field types) to guide follow-up mutations. The guidance is given in two aspects, The first is to enlarge coverage in a cost-effective fashion. For instance, ProFuzzer can avoid fuzzing raw data fields and focuses on mutating fields that determine the interpretation of the subsequent data This would allow us to reach code regions that was otherwise difficult to reach. In the second aspect, the field information can be used to guide exploit generation. For instance, buffer size fields are the root cause of many vulnerabilities. It is known that certain mutation patterns are critical for size fields.

Execution Engine and Report Engine. We use the standard AFL execution and report engines. Each time the probing engine or the mutation engine requests to execute the application, the execution engine forks a new process. During execution, it automatically collects execution profile. The report engine monitors the execution especially for crashes or hangs.

Assumptions. We assume that the target application has the following properties. First, its execution is deterministic: that is, given the same input, multiple executions of the application all follow the same execution path and yield the same result. Second, initial valid seed inputs of reasonable size are available. The focus of ProFuzzer is on type probing and type-based mutation, not on seed selection. Existing seed input generation/selection techniques [27] are hence complementary to ProFuzzer. Third, if the validation on certain bytes fails, the execution will quickly terminate, which means that an exceptional execution has a shorter execution path than a normal one. We have performed an empirical study in Section V-A to validate the aforementioned assumptions.

IV. Design and Implementation

Fuzzing Related Input Field Types. As mentioned earlier, knowledge about application-agnostic input field types can provide strong guidance for fuzzing. For this purpose, we identify 6 input field types, including *assertion*, *raw data*, *enumeration*, *loop count*, *size* and *offset*. These types cover most input content for popular applications, such as image processing tools, document readers and compression utilities. Most importantly, these types affect an program's execution in a unique way and therefore are valuable for enhancing the effectiveness of fuzzing on the program. In the following discussion of these types, we use Figure 1a as an example. For each type, we present the simplified code snippet of the BMP conversion component in OpenJPEG for processing the inputs of that type. Later we will show how to identify these types and how to utilize them during fuzzing.

• *Assertion*. An input field of the assertion type has only a single valid value that allows the program to execute correctly. Other values will result in program termination with errors. The bytes at offsets $0x00$ and $0x01$ in the example image form an assertion field. As shown in the following code snippet, OpenJPEG checks the field to determine whether it equals to $0x4d42$, which is the magic number of a BMP file. During fuzzing, the content of any assertion field should not be changed. Otherwise, the test case will become invalid.

```
header.bfType = get2Bytes(IN);
if (header.bfType != 0x4d42) exit_error();           (i)
```

• *Raw Data*. For an input field of the raw data type, its content does not affect the execution of a program (e.g., its control flow and memory accesses). Note that raw data are implementation specific. The bytes from offset $0x26$ to $0x2d$ in our example image are image resolution, which is important for an image rendering application. But they are raw data for OpenJPEG because it only converts formats without rendering images. We do not need to mutate raw data fields during fuzzing.

• *Enumeration*. An input field of the enumeration type is characterized by a small set of valid values, with other values causing the program to erroneously terminate. An example is the field from offset $0x1c$ to $0x1d$ in the example image. These bytes represent an image's color depth, whose possible values are $0x01$, $0x04$, $0x08$, $0x10$, $0x18$ and $0x20$, according to the file format specification. However, in the current implementation of OpenJPEG, only $0x08$, $0x10$, $0x18$ and $0x20$ are supported. Different processing functions will be invoked to handle different valid color depths, as shown in the following code snippet. In fuzzing, we want to test only the valid values.

```
header.biBitCount = get2Bytes(IN);
switch (header.biBitCount) {
  case 0x08: bmp8toimage(pDataa, ...); break;
  case 0x10: bmp16toimage(pData, ...); break;
  case 0x18: bmp24toimage(pData, ...); break;    (ii)
  case 0x20: bmp32toimage(pData, ...); break;
  default: exit_error();
}
```

• *Loop Count*. An input field of the loop count type determines the times a code fragment within a loopy structure (e.g., for/while loop or recursive invocation) should be executed. Loop count values have substantial impact on path counts and negligible impact on the paths themselves. The byte at offset $0x36$ in the example image is an instance of loop count. It is a raw data field according to the file format specification. However, it denotes the RGB value such that after color transformations (e.g., modified census and discrete wavelet transformations), its value affects the variable bpno as shown in the following code snippet. The variable in turn decides the number of bits that need to be encoded in the loop.

```
data = fread_data(IN);
bpno = color_transformations(data);
for (passno = 0; bpno >= 0; ++passno)             (iii)
    { ...   bpno--; }
```

• *Offset*. An input field of the offset type determines the location from which the program can access the subsequent data in the input file. Offset fields are important for fuzzing as an erroneous offset value may cause the program to access the data outside scope (e.g., the end of file), leading to termination with errors on subsequent validation checks. Within the valid range, different offsets result in different subsequent data being accessed. If the accessed data have impact on the control flow (i.e., non-raw-data), its different values may cause execution to take different paths. As an example, the field from offset $0x0a$ to $0x0d$ in the example image is an instance of offset, which points to the position from which the program loads the image data. In our case, it starts from the 54'th byte of the file. As shown in the following code snippet, if the offset is set to be greater than 54, `fseek()` goes beyond the end of file, causing exception. During fuzzing, an offset field needs to be changed in a way consistent with the preceding position shifting mutations (e.g., adding data fields).

```
header.bfOffBits  = get4Bytes(IN);
fseek(IN, header.bfOffBits, SEEK_SET);          (iv)
if (fread(pData, ..., stride * header.biHeight, IN)
    != (stride * height)) exit_error();
```

• *Size*. An input field of the size type determines the amount of data the program should read from the input file for further processing. Similar to the offset type, if a size value causes any out-of-range data access, the program terminates with errors; within the valid range, different sizes may result in different execution paths, since different data are processed. The difference between offset and size lies in that if a size value is set to zero, errors must occur, while setting an offset to 0 likely does not cause similar problems. The bytes from $0x16$ to $0x19$ in the example image are an instance of size, which represent the image's height. The right size is $0x02$. If the size is set to zero or greater than $0x02$ (e.g., $0x03$), the program terminates with errors. During fuzzing, size field mutation is bounded by the overall input size.

Note that these field types are different from traditional types in programming languages or programmer-defined types. The important design criteria of our type system include *application agnostic* and *fuzzing related*. The six types may not be comprehensive. It is possible that input bytes do not belong to any of them. In such cases, ProFuzzer falls back to the regular random mutation by AFL.

A. Type Probing

Field types are automatically discovered by ProFuzzer via byte-wise mutations and analyzing the impacts of individual mutated bytes on the program execution. Specifically, given a program, ProFuzzer changes its input one byte at a time during probing, enumerating all 256 values of the byte to collect the corresponding 256 execution profile. These profiles are stored in a compact hash array, which keeps the execution frequencies of individual control-flow edges (i.e., control transfer from a basic block to another). We call this representation of runtime profile *edge vector*. Each edge vector is used in our system to extract a number of features that characterize the byte under probing. These features are then utilized to group multiple bytes together to a field and further determine the field's type.

Definitions. To facilitate discussion, we introduce the following definitions. Let I be the input – a sequence of len bytes, each ranging from $0x00$ to $0xff$, and T_I be the output of the function $execute$, which runs the target program on I and returns an edge vector. We have:

$$I = \{0x00, 0x01, ..., 0xff\}^{len}, T_I = execute(I) \tag{1}$$

Let $I_{[i]:v}$ be the result of function $substitute$, which replaces the i-th byte of I with value v.

$$I_{[i]:v} = substitute(I, i, v) \tag{2}$$

For an input byte at offset i that is mutated to value v, we define two metrics based on edge vectors T_I and $T_{I[i]:v}$: *coverage similarity* $S_{I[i]:v}$ and *frequency difference* $D_{I[i]:v}$. The former focuses on measuring the edge coverage similarity (i.e., without considering the number of times an edge is taken), while the latter measures if a dominant number of edges have different execution frequencies. In particular, $S_{I[i]:v}$ is calculated as the size of the edge coverage intersection of the two vectors, divided by the size of their union. $D_{I[i]:v}$ is calculated as the ratio between the number of edges with different frequencies and the number of edges that have different coverage (i.e., covered in one but not the other).

$$\begin{aligned} S_{I[i]:v} &= \frac{\left|\{j |\ T_I[j] * T_{I[i]:v}[j] \neq 0\}\right|}{\left|\{j |\ T_I[j] + T_{I[i]:v}[j] \neq 0\}\right|} \\ D_{I[i]:v} &= \frac{\left|\{j |\ T_I[j] \neq T_{I[i]:v}[j] \wedge T_I[j] * T_{I[i]:v}[j] \neq 0\}\right|}{\left|\{j |\ T_I[j] \neq T_{I[i]:v}[j] \wedge T_I[j] * T_{I[i]:v}[j] = 0\}\right|} \end{aligned} \tag{3}$$

As an example, in the following, we show the edge vectors T_I and $T_{I[0x1c]:0x20}$ for our example image input, whose corresponding inputs differ at offset $0x1c$ with the original value $0x18$ and the mutated value $0x20$. We can see that the two vectors have the same coverage at the edge with index 13 and different coverage at edges 583 and 13052. For edge 34093, both executions cover the edge, but the frequencies are different. The size of edge coverage intersection and union are 1727 and 1766, respectively. Hence, the coverage similarity is 0.978 (1727/1766). The numbers of different edge frequencies and different coverage are 1 and 39, respectively. Hence, the frequency difference is 0.026 (1/39). The two metrics indicate that the two executions are very similar and the difference mainly lies in edge coverage rather than edge frequency.

$$\begin{array}{rcccccccccc} & & & 13 & & 583 & & 13052 & & 34093 & \\ T_I: & < & ... & 1 & ... & 0 & ... & 1 & ... & 4 & ... > \\ T_{I[0x1c]:0x20}: & < & ... & 1 & ... & 1 & ... & 0 & ... & 8 & ... > \end{array}$$

STEP I: Feature Extraction. We extract two features for each byte i as two tuples, each with size 256: F_i^S representing the coverage similarities for mutated values in the range of $[0x00, 0xff]$, called the *coverage feature*; F_i^D representing the frequency differences for the 256 mutated values, called the *frequency feature*.

$$\begin{aligned} F_i^S &= \langle S_{I[i]:0x00}, ..., S_{I[i]:0xff} \rangle \\ F_i^D &= \langle D_{I[i]:0x00}, ..., D_{I[i]:0xff} \rangle \end{aligned} \tag{4}$$

To measure the central tendency of a coverage feature F_i^S, we define its midrange value α_i as:

$$\alpha_i = \frac{max(F_i^S[\cdot]) + min(F_i^S[\cdot])}{2} \quad (5)$$

For easy understanding, we often visualize a coverage feature F_i^S as a bar chart, e.g., Figure 3. The x axis represents the byte value from $0x00$ to $0xff$. The y axis represents the coverage similarity metric corresponding to the byte value. We refer the readers to Appendix A for the coverage feature of each byte in the sample image.

STEP II: Field Identification. The goal of field identification is to group consecutive bytes of the same input type together as a field. Observe that the target program tend to perform validation check on a whole input field instead of individual bytes. Intuitively, if consecutive bytes belong to a same field, it is very likely that they share a large part of their mutation features that correspond to invalid (mutated) values. Specifically, although each byte in a field may have multiple valid values (e.g., an enumeration field) which often lead to various execution paths and hence different metrics, the many invalid values always lead to the same termination path with an exception, which is the shortest path. As such, these invalid values must lead to the same coverage and frequency metrics. Hence, we group bytes at offsets from i to j together as a field, if condition (6) is satisfied.

$$\forall x, y \in [i, j] : min(F_x^S[\cdot]) = min(F_y^S[\cdot]) \quad (6)$$

Note that the minimal coverage similarity value corresponds to the shortest path caused by invalid mutation. Consider the bytes at offsets $0x1c$ and $0x1d$ in our example. The coverage feature of the byte at offset $0x1c$ is shown in Figure 3c. Although the byte may have multiple valid values with high coverage similarity with the original execution (e.g., value $0x20$ having 0.978 similarity and value $0x10$ having 0.959 similarity), when given an invalid value its minimum coverage similarity is 0.068, which is the same as the minimum similarity of the byte at $0x1d$. Recall that the source code has a switch case statement on these two bytes such that all invalid mutations to either byte lead to the same failing path.

STEP III: Field Type Identification. Given a group of consecutive bytes as an input field, the next step is to determine the type of the field.

• *Assertion Fields.* For a field at offsets from i to j, if it satisfies the condition (7), we consider it an `assertion` field. The condition means that for any byte at offset $x \in [i, j]$, there exists one and only one value v for the byte that induces a similarity score 1. And for any other value, the similarity score is less than the midrange value.

$$\begin{gathered} \forall x \in [i, j] : \exists v \in [0x00, 0xff] : F_x^S[v] = 1; \\ \forall u \neq v \in [0x00, 0xff] : F_x^S[u] < \alpha_x; \end{gathered} \quad (7)$$

The intuition is that any byte in an assertion field has only one valid value that allows the program to proceed normally. Other values will result in the same termination-with-exception path, leading to low similarity with the original execution. The midrange value α_x can be used to distinguish the original execution and erroneous executions.

For example, the field at offsets $[0x00, 0x01]$ in Figure 1a are an assertion field. Figure 3a visualizes the similarity feature of the byte at $0x00$. As we can see, only the value of $0x42$ results in a normal execution with the similarity score 1, other values result in an exception with the similarity score 0.059.

• *Raw Data Fields.* For a field at offsets from i to j, if it satisfies the condition (8), we consider it a `raw data` field. The condition means that for any byte with offset $x \in [i, j]$, its similarity score compared with the original execution is always 1 for all values.

$$\forall x \in [i, j], \forall v \in [0x00, 0xff] : F_x^S[v] = 1 \quad (8)$$

The intuition is that the value of a raw data field must not affect the control flow of the program. For example, the field in offsets $[0x26, 0x2d]$ in Figure 1a are treated as a raw data field. It denotes the image resolution, which has no impact on control flow of image conversion. Figure 3b visualizes the similarity feature of byte 0x26.

• *Enumeration Fields.* An enumeration field satisfies condition (9), which means that there exists a byte at offset $x \in [i, j]$ and a set VS which is a proper subset of the set $\{0x00, ..., 0xff\}$ with size larger than 1, such that for any value v in VS, its similarity score is larger than the midrange value and for any byte u not in VS, its similarity score is smaller than the midrange value.

$$\begin{gathered} \exists x \in [i, j], \exists VS \subset \{0x00, ..., 0xff\}, |VS| > 1 : \\ \forall v \in VS : F_x^S[v] > \alpha_x; \forall u \notin VS : F_x^S[u] < \alpha_x; \end{gathered} \quad (9)$$

The intuition is that an enumeration field has a small set of valid values. For those valid values, the similarity scores are relatively high, since the program can correctly proceed. For other values, however, the similarity scores are very low, since the control flow most likely transfers to error handling code.

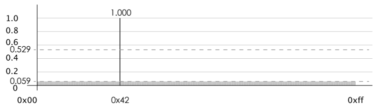
The field at $[0x1c, 0x1d]$ in Figure 1a is an enumeration field (indicating color depth). Figure 3c visualizes the coverage feature of byte 0x1c. The figure also tells us that the current implementation of OpenJPEG supports four image formats.

• *Loop Count Fields.* A loop count field satisfies the condition (10), meaning that the variance of the coverage similarity scores is less than β, which is a predefined value indicating small data variation [28], [29]. The average of frequency difference is larger than 1, which means frequency difference is dominating when compared with coverage difference.

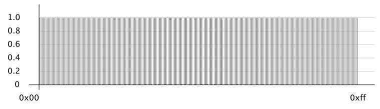
$$\exists x \in [i, j] : Variance(F_x^S[\cdot]) < \beta, Average(F_x^D[\cdot]) > 1 \quad (10)$$

The intuition behind this condition is that a loop count value has substantial impact on edge frequency and negligible impact on edge coverage. The byte at offset 0x36 in Figure 1a is a loop count field. It decides the number of bits that need to be encoded in a loop. Figure 3d visualizes the coverage feature of the byte. We can see that the similarity scores of the values are very close to each other, which means they have little difference in coverage. But when we calculate the ratio between the number of edges with different frequencies and the number of edges with different coverage, the mean value is 7.355, indicating the dominance of the former.

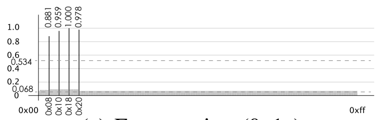
• *Offset Fields.* An offset field needs to satisfy (11), which means that there exists a byte at offset $x \in [i, j]$ such that



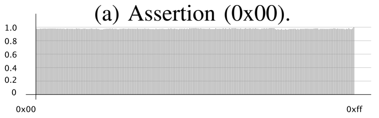
(a) Assertion (0x00).



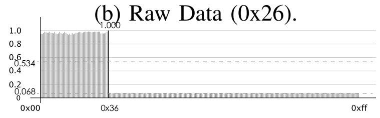
(b) Raw Data (0x26).



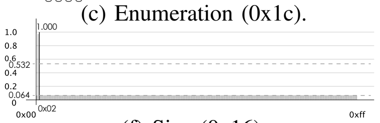
(c) Enumeration (0x1c).



(d) Loop Count (0x36).



(e) Offset (0x0a).



(f) Size (0x16).

Fig. 3: Similarity features of each input field type.

the coverage similarity score of value 0 (i.e., the coverage similarity with the original execution when the byte is mutated to 0) is larger than the midrange, and there also exists a value t in range $[0x00, 0xff]$, such that there are two values u and v smaller than t have different similarity scores, while any value w larger than t always has a similarity score less than the midrange value.

$$\begin{aligned}&\exists x \in [i,j] : F_x^S[0] > \alpha_x; \exists t \in [0x00, 0xff] :\\ &\exists u, v \in [0,t] : F_x^S[u] \neq F_x^S[v]; \forall w \in (t, 0xff] : F_x^S[w] < \alpha_x\end{aligned} \tag{11}$$

The intuition is that the valid range of an offset field is bounded by the length of the remaining input (i.e., from the byte next to the field to the end). Mutating the field to any value larger than the bound may cause out of range access and hence exception. Mutating the field to any value smaller than the bound results in different subsequent data being accessed, which in turn may induce different execution paths. As such, there are two such values u and v whose coverage similarity metrics are different. In addition, mutating the field to 0 is very likely valid as that means the program would access subsequent data starting right after the field.

For example, the bytes at $[0x0a, 0x0d]$ in Figure 1a are an offset field that points to a position from which the program loads the image data. Figure 3e visualizes the coverage feature of byte $0x0a$. We can see that $0x36$ is the bound. Beyond the bound, the coverage similarity metrics are low as the program terminates with errors, leading to substantial differences from the original execution. Within the boundary, different offset values result in different similarity metrics. The coverage similarity for value 0 is larger than the midrange.

• *Size Fields.* A size field needs to satisfy a condition similar to (11) except that there exists a byte at offset $x \in [i,j]$ such that the coverage similarity metric for value 0 is smaller than the midrange. Intuitively, a size field has similar characteristics to an offset field as there also exists a bound determined by the length of the remaining input. The difference is that mutating an offset field to 0 likely leads to successful execution whereas mutating a size field to 0 often leads to early termination.

For example, the bytes at $[0x16, 0x19]$ in Figure 1a are a size field that indicates the image height. Figure 3f visualizes the coverage feature of byte $0x16$. Note that value $0x02$ is the bound as we use a small image as input. The values larger than the bound have a very small similarity score, indicating early termination and the score for value 0 is below the midrange.

We distinguish size fields from offset fields as they have different mutation policies. For example, when increasing an offset field by 1, we insert a byte right after the field so that the same original subsequent data is accessed in the mutated run. In contrast, when increasing a size field by 1, we insert a byte at the end of input. In the case that the input ends with an assertion field, the byte is added before the field.

In some cases, an offset/size field may be next to a raw data field. Identifying the boundary of these fields could be tricky because mutating the least byte of an offset/size field may not cause any execution changes, making it indistinguishable from a raw data byte. We handle the problem by grouping the identified size/offset bytes with their adjacent raw bytes as long as the value denoted by the grouped bytes does not exceed the size of the seed input.

Our experiment shows that our field probing is highly precise, with only 5.2% false positive rate and 4.5% false negative rate on average (see Section V-C).

B. Reprobing

During fuzzing, given a new (mutated) input, ProFuzzer first tries to reuse a probed template whose execution trace is similar to the execution trace of the input. If there is not such a template, we consider the mutated input to be a new seed input and re-probe its fields just like probing of the original seed input. Before reprobing, ProFuzzer leverages the AFL built-in input trimming functionality to reduce the input size as much as possible and prioritize the inputs with small sizes.

The challenge lies in that the new seed input may not lead to a normal execution, but rather an execution with exception. Therefore, the basic idea of reprobing is *when mutating the new seed input results in a longer execution than the seed, we consider the reprobing mutation is getting closer to a valid input, and hence we replace the current seed with the mutated seed and continue*. Note that a longer execution indicates the mutated seed gets through more validation checks.

Reprobing Feature Extraction. Due to the invalidity of seed executions, reprobing has to be based on different execution feature that measures execution length variation. For a byte at offset i that is mutated to a value v, we define a reprobing metric $R_{I[i]:v}$ based on edge vectors T_I and $T_{I[i]:v}$. The metric measures the ratio between the number of edges taken during the executions, that is, the number of edges with a non-zero count in T_I and that in $T_{I[i]:v}$. Then, we extract the *reprobing feature* F_i^R for a byte at offset i as a tuple of 256 elements, each of which corresponds to a reprobing metric.

$$\begin{aligned}R_{I[i]:v} &= \frac{|\{j |\ T_{I[i]:v}[j] \neq 0\}|}{|\{j |\ T_I[j] \neq 0\}|}\\ F_i^R &= \langle R_{I[i]:0x00}, ..., R_{I[i]:0xff}\rangle\end{aligned} \tag{12}$$

Field Type Re-Identification. In the following, we explain how the new feature is used to reprobe assertion fields and enumeration fields. Reprobing other fields is elided due to space limitations.

• *Reprobing Assertion Fields.* For a byte at i, if it satisfies condition (13), we consider it an assertion field. The condition means that there exists one and only one value v for the byte that induces longer execution than the original execution. And for any other value, the execution is shorter than or equal to the original execution. After identifying such a value, we replace the byte with the identified value to allow the execution to proceed farther. Multiple such bytes are grouped into an assertion field following the same method described in Section IV-A.

$$\begin{aligned} &\exists v \in [0x00, 0xff] : F_i^R[v] > 1; \\ &\forall u \neq v \in [0x00, 0xff] : F_i^R[u] \leq 1; \end{aligned} \tag{13}$$

• *Reprobing Enumeration Fields.* For a byte at offset i, if it satisfies condition (14), we consider it a byte belonging to an enumeration field. The condition means that there exists a set VS which is a proper subset of the set $\{0x00, ..., 0xff\}$ with size larger than 1, such that for any value v in VS, the resulted execution is longer than the original execution. And for any other value, the execution is shorter than or equal to the original one. Multiple such bytes can be grouped to an enumeration field.

$$\begin{aligned} &\exists VS \subset \{0x00, ..., 0xff\}, |VS| > 1 : \\ &\forall v \in VS : F_i^R[v] > 1; \forall u \notin VS : F_i^R[u] \leq 1; \end{aligned} \tag{14}$$

C. Type Guided Mutation

Probed fields are used to guide mutation. The guidance is provided in two aspects. In the first aspect, it limits the mutation to all the legitimate values of the field type to achieve better coverage. We call it *exploration mutation*. In the second aspect, ProFuzzer mutates each field to exploit a set of special values (for the specific field type) that could lead to potential attacks. We call it *exploitation mutation*. During fuzzing, ProFuzzer interleaves these two modes. Intuitively, it tries to exploit the paths covered by the current seeds. If the process becomes fruitless, it starts to explore more coverage by generating more seeds, and so on.

Exploration Mutation. An important advantage of having field information is that we can apply mutations at the field level instead of the byte level, which substantially reduces the number of ineffective mutations. Besides, we applying the following mutation policies in the exploration stage. (1) We do not allow any mutation on raw data fields. (2) For an enumeration field, we assign a value within its valid range (see Section IV-A) with a high probability (0.9 in our research) and a value out of the range with a small probability (0.1) to explore other valid values missed during probing. In particular, we leverage a technique similar to VUzzer [12] to extract constants from subject programs (e.g., numbers and strings) such that those values of the same size as the field are mutation candidates. (3) Even for an assertion field, which we are not supposed to change its content during fuzzing, we still modify it using extracted constants with a small probability (0.1) in case probing misclassifies the field (e.g., treating an enumeration field as an assertion). (4) For an offset field, when mutation increases (or decreases) its value by X, we insert (or delete) X bytes right after the field. (5) For a size field, when mutation increases (or decreases) its value by X, we insert (or delete) X bytes before the last non-assertion field at the end. The inserted data is randomly generated. (6) For a loop count field, we perform a binary search to identify its minimum and maximum valid values. (7) For other fields that cannot be classified into one of the six types, we perform the default AFL random per-byte mutation.

TABLE I: Exploitation rules.

Type	Rules
Size	01. $[size] = max(\{[field] \mid field \in SizeField\})$
	02. $[size] = min(\{[field] \mid field \in SizeField\})$
	03. $[size] = max(\{len(field) \mid field \in RawDataField\})$
	04. $[size] = min(\{len(field) \mid field \in RawDataField\})$
	05. $[size] = location\_end(I) - location\_current(I)$
	06. $[size] = value \in BoundaryValue$
Offset	07. $[offset] = max(\{[field] \mid field \in OffsetField\})$
	08. $[offset] = min(\{[field] \mid field \in OffsetField\})$
	09. $[offset] = location\_end(I) - location\_current(I)$
	10. $[offset] = location\_start(I)$
	11. $[offset] = location\_current(I)$
	12. $[offset] = location\_end(I)$
	13. $[offset] = value \in BoundaryValue$
Loop Count	14. $[count] = max(\{[field] \mid field \in SizeField\})$
	15. $[count] = max(\{[field] \mid field \in OffsetField\})$
	16. $[count] = max(\{len(field) \mid field \in RawDataField\})$
	17. $[count] = value \in BoundaryValue$

Exploitation Mutation. We manually analyzed more than a hundred of PoCs of known vulnerabilities and found that a given field type often has a small set of values that could lead to exploits. Such patterns are application agnostic. Since we have recognized fields and field types, exploiting these patterns helps effectively and efficiently discover new vulnerabilities. Particularly, for each PoC, we used ProFuzzer to probe its field template and then conducted a manual study of field types and the corresponding values that lead to exploitation. Currently, the procedure requires domain knowledge and manual efforts. We leave the automatic learning of exploitation rules from a larger pool of PoCs to our future work.

Table I presents our findings for size, offset, and loop count fields. The others are elided due to space limitations. Consider Rule 1 which specifies that we may change a size field to the max value of any size fields ever found for the program. Intuitively, various size values correspond to different data structures whose application-specific types we do not care. The mutation essentially forces the program to interpret the subsequent data as a different data structure (with a larger size). If the program has type related bugs such as type confusion [30], the mutation would help to expose the problems. Rules 2, 3, 4, 7, 8 have a similar idea. Rule 9 means that by the specified mutation, we force the program to start reading beyond the end of input. Note that during exploitation, our goal is to expose bugs. As such, when increasing an offset field, we do not need to patch the input with new bytes.

V. Evaluation

Target Programs. We use 40 popular real-world programs to study the generality of our assumptions, and select 10 of them that are commonly used in existing fuzzing projects for evaluation. The selected programs cover different application categories including image processing, audio and video decoder, PDF reader, and compression library, which have been used to evaluate other fuzzers [31], [32]. We also compare ProFuzzer with other fuzzers on standard benchmarks.

Computing Resource. We divide our tasks into two machines. The first one is used for discovering zero-day vulnerabilities. It has 32-cores (Intel® Xeon^{TM} CPU E5-2690 @ 2.90GHz) with 256G main memory. And the other one runs experiments to compare ProFuzzer with other fuzzers. The machine has 24-cores (Intel® Xeon^{TM} CPU E5-2630 @ 2.30GHz) and 64G main memory. Each fuzzing task only takes one CPU core, except Driller and QSYM, whose tasks use two CPU cores, one for fuzzing, and the other for symbolic execution.

ASAN Setting. In our evaluation, ASAN was enabled for finding zero-day vulnerabilities (Section V-D), which is a common practice in real-world testing, and for exposing known vulnerabilities (Section V-F), since some of them can only be detected with ASAN. For each target program, two configurations were used: the ASAN-disabled version for fuzzing and the ASAN-enabled version for checking whether a vulnerability is triggered on the generated test case. The execution of ASAN-enabled version is on a separate process and hence does not affect the fuzzing runtime. When evaluating the performance of ProFuzzer (Section V-G) and comparing with other fuzzers on standard benchmarks (Section V-E), ASAN was disabled.

A. Generality of Assumptions

We randomly sample 40 popular programs used in AFL [33] and OSS Fuzz [32] to study the generality of our assumptions about (1) deterministic execution, (2) the size of initial seed inputs, and (3) the difference between normal execution and exceptional execution. For each application, we collect its initial seed inputs from its test-suite or public seed corpus [34], [35]. We run each application multiple times to check whether its execution is deterministic. We also conduct statistical analysis on the edge count of normal execution and exceptional execution. The detailed results are presented in Appendix B. Overall, all the sampled applications are deterministic and have initial seed inputs of moderate size (ranging from 78 to 1650 bytes). The difference between normal and exceptional executions is significant, with the edge coverage of the latter 4 times smaller on average.

B. Input Size and Path Coverage

Input Size Distribution. The cost of probing/reprobing is proportional to input size. ProFuzzer has a few internal mechanisms that allow controlling input sizes (without affecting fuzzing effectiveness). The figure in Appendix C(a) describes the input size distribution of OpenJPEG. For comparison, the box on the left illustrates the "interesting inputs" found by vanilla AFL, which contributed to new coverage. The size ranges from 4 to 914 bytes. The right shows the sizes of inputs by ProFuzzer, which largely range from 60 to 80 bytes

TABLE II: Probing accuracy.

Product	Actual	ProFuzzer			afl-analyze		
		Inferred	Wrong (FP*)	Missed (FN**)	Inferred	Wrong (FP*)	Missed (FN**)
exiv2	20	21	3 (14%)	0 (0%)	16	11 (69%)	15 (75%)
graphicsmagick	17	19	1 (5%)	2 (12%)	7	4 (57%)	14 (82%)
libtiff	20	23	2 (9%)	3 (15%)	17	9 (53%)	12 (60%)
openjpeg	17	17	1 (6%)	0 (0%)	9	4 (44%)	12 (71%)
libav	14	14	1 (7%)	0 (0%)	4	2 (50%)	12 (86%)
libming	14	14	0 (0%)	0 (0%)	3	1 (33%)	12 (86%)
mupdf	52	53	2 (4%)	1 (2%)	34	13 (38%)	31 (60%)
podofo	52	53	1 (2%)	2 (4%)	25	11 (44%)	38 (73%)
lrzip	39	39	0 (0%)	5 (13%)	30	3 (10%)	12 (31%)
zziplib	36	36	2 (6%)	0 (0%)	14	4 (29%)	26 (72%)

[*] FP = #Wrong / #Inferred [**] FN = #Missed / #Actual

with a few outliers more than 350. Even with the smaller inputs, our approach achieves 93% more path coverage than AFL, thanks to our strategies of prioritizing small inputs and reusing templates when possible (Section V-G). Specifically, we first probe new inputs with smaller sizes. For each input, ProFuzzer first checks its trace and reuses the template of a similar execution when possible. In addition, we use the AFL built-in input trimming functionality to reduce the input size.

Input Size versus Path Coverage. To understand the relation between input size and path coverage, we divide the generated test inputs into different groups based on their sizes, and calculate the average path coverage of each group. The figure in Appendix C(b) shows the result for OpenJPEG. When the size is less than 80, the increase of input size improves path coverage. However, coverage degrades when the size exceeds 80. Two factors lead to the coverage decrease when the size increases: 1) the larger the input, the more time each execution takes; 2) the larger the input, the more mutations and time are needed for validating such mutations. Similar results are observed on other applications. This supports our strategy of prioritizing small inputs.

C. Probing Accuracy

The experiment is to measure whether ProFuzzer can correctly identify fields and their types through probing. We acquire the ground truth by manually checking how applications handle individual bytes, and then compare the probing results with the ground truth to measure false positives (wrong fields) and false negatives (fields missed) by ProFuzzer. We also compare ProFuzzer with AFL-analyze [19], a simple off-line file format analysis utility included in AFL, which works by performing four kinds of mutations for each byte (i.e., xor with $0xff$ and $0xfe$, add and sub by 10), and classifies its data type based on simple heuristics (e.g., a size field is identified if all the four mutations yield different traces from the original execution and the value is a small integer). It only recognizes raw data, assertion, and length types.

Table II shows the results. Observe that ProFuzzer has very few false positives (5.3% on average) and false negatives (4.6% on average), while AFL-analyze has a relatively high FPs (42.7% on average) and FNs (69.6% on average). Among the 276 fields correctly recognized by ProFuzzer, only 97 of them are identified by AFL-analyze. This illustrates the simple approach by AFL-analyze is ineffective. A close look at ProFuzzer results shows that it may misclassify a field when the runtime characteristics of different field types are not clearly distinguishable during probing. For example, `lrzip` computes a checksum on the compressed data. ProFuzzer decides that the compressed data is a large *assertion* field as any change to any single byte causes exception, whereas the correct type shall be *raw data*.

D. Finding Zero-day Vulnerabilities

Summary. Table III summarizes the results of using ProFuzzer to find zero-day vulnerabilities. The ProFuzzer prototype has run for 2 months before writing the paper. It has found 42 unreported vulnerabilities (column 3); 30 of them have been assigned a CVE number, and 27 of them have been confirmed or fixed by the vendors after our reports. Our subject programs are all widely used, actively maintained and have been well tested and fuzzed by the developers and security researchers [17], [36]–[38]. The fact that ProFuzzer is capable of finding these new vulnerabilities in such a short period of time demonstrates the effectiveness of our design. The full list of the zero-day vulnerabilities can be found in Appendix D.

Case Study. We take the discovery of the zero-day vulnerability CVE-2017-17725 as an example to demonstrate the effectiveness of ProFuzzer. It is a buffer over-read in Exiv2, which may cause information leak to facilitate further exploitation (e.g., bypass address space layout randomization protection). Exiv2 is a popular open-source image metadata management library, which is used in many projects including KDE and Gnome Desktop. The vulnerability can affect the downstream products depending on the Exiv2 library. Although Exiv2 has been extensively tested, this vulnerability still fell through the crack. We reported this vulnerability to Exiv2 vendor. They acknowledged our finding and fixed it in 3 days.

Figure 4a shows the vulnerable code. The function `readMetadate()` allocates a `subBox.length+8` bytes buffer and stores the pointer to `data.pData_` (line 273). Then it calls function `getULong()` to read 4 bytes from `data.pData_` at the offset 3. The size of the buffer should be larger than 7 to avoid over-read. The variable `subBox.length` (line 273) is read from a certain size field in the input file. If the input file size is in the range $[0xfffffff8, 0xfffffffe]$, after 8 is added, there will be an integer overflow making the allocated buffer smaller than 7 bytes, resulting in buffer over-read.

We investigated the mutation history and found that field information is critical. Figure 4b shows a mutated input. The bytes at the offsets from $0x3e$ to $0x41$ are loaded as the value of `subBox.length`. ProFuzzer probes that these four bytes constitute a size field. In the exploration mutation, it mutates the whole field rather than individual bytes and finally triggers the vulnerability when mutating the size field to be a value within $[0xfffffff8, 0xfffffffe]$. In contrast, it is very difficult for AFL to discover this case because values in the range $[0xfffffff8, 0xfffffffe]$ are not common boundary values. Furthermore, since AFL does not know which consecutive bytes form a field, it randomly selects consecutive subsequences. The search space is hence massive.

TABLE III: Vulnerabilities discovered by ProFuzzer.

Category	Product	SLOC	Bugs	CVEs	Fixes
Image	exiv2	131,993	5	5	5
	graphicsmagick	299,186	2	1	1
	libtiff	82,484	8	1	1
	openjpeg	164,284	3	3	3
Audio & Video	libav	703,369	3	2	0
	libming	72,747	2	2	2
PDF	mupdf	102,824	1	1	1
	podofo	78,195	6	6	3
Compression	lrzip	19,098	3	3	3
	zziplib	12,898	8	6	8
Total	*10*	*1,667,078*	*42*	*30*	*27*

```
File: src/jp2image.cpp
206 void Jp2Image::readMetadata() {
273   DataBuf data(subBox.length+8);
274   io_->read(data.pData_,data.size_);
275   long iccLength = getULong(data.pData_+3, ...);
442 }
                                                              (v)
File: src/types.cpp
243 uint32_t getULong(const byte* buf, ...) {
250   return (byte)buf[0] << 24 | (byte)buf[1] << 16
251          | (byte)buf[2] <<  8 | (byte)buf[3];
253 }
```

(a) Vulnerable code.

```
           0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000030h  00 00 00 01 00 00 00 01 00 03 07 07 00 00 00 00
00000040h  00 0F 63 6F 6C 72 01 00 63 FF 4F FF 51 00 2F 00
00000050h  00 00 00 00 0A
                                          FF FF FF FB
```

(b) Mutated input.

Fig. 4: Discovery of CVE-2017-17725.

E. Evaluation on Standard Benchmarks

Benchmarks. LAVA-M dataset [15] and Google fuzzer test-suite [16] are two real-world standard benchmarks for fuzzer evaluation. The LAVA-M dataset includes four utilities from `coreutils`, each of which is injected with crafted bugs. The Google fuzzer test-suite contains real-world libraries that have interesting bugs and hard-to-find code paths.

Fuzzers. We compare ProFuzzer with AFL and its optimized version AFLFast [7], as well as three state-of-the-art fuzzers Driller [13], VUzzer [12] and QSYM [14]. AFLFast accelerates AFL by prioritizing low-frequency paths to achieve higher path coverage. Driller enhances AFL through symbolic execution whenever the fuzzer gets stuck at complicated path conditions. VUzzer utilizes dynamic tainting to locate interesting bytes (e.g., those related to a certain predicate), and mutate them to constant values extracted from the binary. QSYM proposes a tailored concolic executor, which is scale to real-world applications. Another related fuzzing tool, Angora [39], which uses gradient descent to guide fuzzing, was not available at the time of writing this paper.

LAVA-M. LAVA-M identifies input bytes that do not affect control flow and then injects buggy code guarded by comparing those bytes to some magic value. Figure 5 shows an example. In line 395, `buff_data` contains data derived from some inputs bytes. The values of the four consecutive bytes are stored to a global variable. Then line 71 checks if it matches a magic value $0x6c617523$. If so, the pointer `fp` is offset to an invalid address, causing a segment fault.

We ran the fuzzers for five hours on each test program with the seed inputs provided by VUzzer and by the Angora

TABLE IV: Bugs found in LAVA-M.

Program	Listed Bugs	Found Bugs										
		ProFuzzer		AFL		AFLFast		Driller		VUzzer	Angora	QSYM
		ori	opt	ori	dict	ori	dict	ori	dict			
base64	44	11	24	2	14	2	16	4	27	18	48	44
md5sum	57	4	10	0	20	0	16	3	14	20	57	52
uniq	28	13	28	0	18	4	13	2	12	26	28	28
who	2136	17	41	7	44	6	48	7	54	47	1541	1071
Total	2265	45	103	9	96	12	93	16	107	111	1674	1195

```
File: src/uniq.c
395 int lava_318 = buff_data[0] << (0*8);
    lava_318 |= buff_data[1] << (1*8);
    lava_318 |= buff_data[2] << (2*8);
    lava_318 |= buff_data[3] << (3*8);                         (vi)
    lava_set(318,lava_318);

File: lib/fclose.c
57 FILE *fp;
71 fileno(fp+(lava_get(318))*(0x6c617523==(lava_get(318)));
```

Fig. 5: An example of injected bug in LAVA-M.

authors in order to faithfully reproduce their results. Table IV shows our findings. Given the nature of the injected bugs, we equipped ProFuzzer with a dictionary (i.e., constants extracted from the binary, which were also used by VUzzer) and further optimized its mutation policy to focus more on the input fields that have little impact on edge coverage. Here we report the results of the original and the optimized ProFuzzer. For AFL, AFLFast and Driller,the results include the situations with and without the dictionary. For Angora, we used the result from the paper since it was not available at the time of writing this paper.

After customizing the mutation policy, the optimized ProFuzzer was found to achieve performance comparable with Driller and VUzzer, when they use dictionaries. Dictionaries are important because of the simple triggering mechanism of the injected bugs, which checks whether certain input bytes equal to a predefined constant. Angora and QSYM have far better performance than other fuzzers on this dataset. We believe that this advantage comes from the nature of the injected bugs that fits their fuzzing strategies very well but may not reflect real-world situations. Appendix E presents the list of bugs found by ProFuzzer.

Google Fuzzer Test-suite. Table V shows the results on the Google fuzzer test-suite that contains more complex programs than LAVA-M. The objective of this test suite is to see if a fuzzer can reach certain code locations, which is more realistic than the injected bugs in LAVA-M as well. The table presents target program (column 1), target locations (column 2) and time taken to reach the location for the five fuzzers (columns 3 to 7). Here, we set the timeout to 24 hours. In summary, ProFuzzer is able to reach 9 out of total 11 locations while other fuzzers can reach only 6, and all locations hit by other fuzzers can also be identified by ProFuzzer. For those that can be reached by ProFuzzer and at least another fuzzer, ProFuzzer takes the least time except for the target location in JSON (row 2) and one of the target locations in libpng (row 7). These two locations are easily to found. ProFuzzer takes more time due to probing. Considering the cases that did not timeout, on average ProFuzzer is 3.51 times faster than AFL, 2.26 times faster than AFLFast. 3.24 times faster than Driller, and 8.55 times faster than VUzzer. ProFuzzer outperforms Driller and VUzzer in more complex programs with more realistic objectives for the following reasons. (1) Driller executes programs in the slow QEMU mode due to the need of collecting symbolic constraints. (2) Each time when the concolic execution is activated, Driller feeds all seeds recorded by the fuzzer to its symbolic execution engine, angr, rendering significant overhead. Many these seeds are rather complex due to the complexity of the programs. (3) VUzzer requires heavyweight dynamic taint analysis.

TABLE V: Evaluation on Google fuzzer test-suite.

Program	Location	Reaching Time (hours)				
		ProFuzzer	AFL	AFLFast	Driller	VUzzer
guetzli	output_image.cc:398	0.83	3.64	2.37	3.73	8.60
json	fuzzer-..._json.cpp:50	0.05	0.02	0.04	0.12	4.26
lcms	cmsintrp.c:642	0.67	6.55	3.83	5.31	11.97
libarchive	archive_..._warc.c:537	1.34	7.88	6.92	6.74	14.42
libjpeg	jdmarker.c:659	11.68	T/O	T/O	T/O	T/O
libpng	png.c:1035	1.84	3.37	2.33	4.27	6.06
	pngread.c:757	0.03	0.01	0.01	0.02	0.17
	pngrutil.c:1393	7.63	T/O	T/O	T/O	T/O
vorbis	codebook.c:479	T/O	T/O	T/O	T/O	T/O
	codebook.c:407	T/O	T/O	T/O	T/O	T/O
	res0.c:690	11.76	T/O	T/O	T/O	T/O

F. Exposing Known Vulnerabilities

We further compare the effectiveness of ProFuzzer with other fuzzers on real-world known vulnerabilities. From each application category, we randomly choose a program, and then manually collect recent vulnerabilities in the programs and run the five fuzzers on them to see if they can disclose all these bugs. As shown in Table VI, ProFuzzer exposed 18 out of 20 known vulnerabilities within 24 hours, while AFL, AFLFast, Driller, VUzzer and QSYM discovered 11, 13, 11, 7 and 15 respectively. On average, ProFuzzer uses 2.63 hours to find a known vulnerability, about 50%, 73%, 35%, 18% and 70% of the time used by AFL, AFLFast, Driller, VUzzer and QSYM respectively. Actually, within 24 hours, among hundreds of test cases produced by QSYM for openjpeg, libming, podofo and zziplib each, 2, 16, 45, 21 new test cases were found to be missed by QSYM's embedding AFL fuzzing engine.

Case Study. We take the known vulnerability CVE-2017-5975 as an example to demonstrate the effectiveness of ProFuzzer. It is a buffer over-read in ZzipLib, which is a lightweight open-source archive library, allowing to extract data from files archived in a single zip file. It is widely used in popular software (e.g., PHP 4) and other libraries (e.g., LibClamav). The vulnerability can affect the downstream software/libraries.

Figure 6a presents the vulnerable code. The function `zzip_mem_entry_extra_block()` is called to search data block of a special data type `ZZIP_EXTRA_zip64` ($0x01$) in the extra data area. The search is implemented in a while loop (lines 247-259). In each iteration, the cursor moves forward with a stride decided by an offset field in the input file (line 256). Once such a block is found, its pointer will be returned to the caller function `zzip_mem_entry_new()` for accessing its `z_csize` field, whose offset from the block header is $0x04$ (line 221). If a mutated input file has the offset field pointing to a location whose value is $0x01$, indicating the

TABLE VI: Time consumption to expose known bugs.

Product	Vulnerability	Exposing Time (hours)					
		ProFuzzer	AFL	AFLFast	Driller	VUzzer	QSYM
openjpeg	Issue#996	0.20	4.26	2.37	3.53	2.32	2.11
	CVE-2016-10504	3.21	6.54	4.82	9.17	T/O	5.02
	CVE-2016-10507	0.22	T/O	T/O	T/O	T/O	T/O
	CVE-2017-12982	2.55	7.13	3.58	10.24	T/O	5.76
	CVE-2017-14151	1.13	T/O	T/O	T/O	T/O	T/O
libming	CVE-2016-9264	0.73	T/O	T/O	T/O	T/O	5.54
	CVE-2016-9266	1.95	0.85	0.97	1.97	3.18	1.03
	CVE-2016-9828	5.02	T/O	T/O	T/O	T/O	7.72
	CVE-2016-9829	1.07	1.20	1.47	2.31	T/O	1.26
	CVE-2017-9989	0.85	0.98	1.18	1.45	1.25	0.92
podofo	CVE-2017-5852	1.57	1.75	1.87	4.08	18.77	1.66
	CVE-2017-5886	10.31	T/O	13.82	T/O	T/O	12.07
	CVE-2017-7378	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2017-8378	T/O	T/O	9.18	T/O	T/O	T/O
	CVE-2017-8787	7.95	T/O	T/O	T/O	T/O	T/O
zziplib	CVE-2017-5975	0.03	0.13	0.13	0.13	0.24	0.10
	CVE-2017-5976	0.03	0.03	0.02	0.11	0.17	0.02
	CVE-2017-5977	8.19	T/O	T/O	T/O	T/O	7.78
	CVE-2017-5978	2.23	4.50	2.80	5.63	T/O	3.06
	CVE-2017-5980	0.11	0.17	0.13	0.72	0.84	0.15

special data type, and the location itself is less than $0x04$ bytes away from the end of the file (i.e., the data block has an empty body), a buffer over-read occurs.

Figure 6b show a mutated input by ProFuzzer. The bytes at $0x52$ and $0x53$ are recognized as an *offset*, which in fact is used in the while loop of block searching. In the exploitation mutation, ProFuzzer applies the Rule 09 presented in Table I to set its value to be the difference of the current location and the end of file location. Such mutation results in significant differences in executions, which in turn trigger reprobing. During reprobing, the byte at offset $0x6d$ is mutated with value ranging from $0x00$ to $0xff$ to collect the features. When mutated to $0x01$, the vulnerability is triggered. We should note that our PoC is smaller than the PoC provided by the original discovery of this vulnerability (111 bytes v.s. 151 bytes).

More interestingly, ProFuzzer discovers a zero-day (CVE-2018-6381) two hours after exposing the known vulnerability. It is a heap buffer over-read that allows 1024 bytes information leak. We investigate ProFuzzer's mutation history and identify a key mutation step. It increases the value of a certain *size* field (file name length) by $0x1c$ and correspondingly appends $0x1c$ bytes of random data to the end of the file following our mutation policies. The appended data is misinterpreted as the content of a `ZZIP_EXTRA_zip64` extra block. As such, random (added) bytes are interpreted as a size field with a large value. Finally, this misinterpreted size value is compared with 1024 and the smaller value is used as the size of the data to be copied in a `memcpy` call. Note that the misinterpreted fields and field types are synthesized by ProFuzzer according to the buggy implementation of the ZzipLib product. It cannot be generated using the standard zip file format specification. We reported this vulnerability to the ZzipLib vendor. They acknowledged our finding and fixed it in 4 days.

```
File: src/zzip/memdisk.c
zzip_mem_entry_new(DISK * disk, ...) {
  block = zzip_mem_entry_extra_block(entry, 0x01);
  entry->zz_usize = __zzip_get64(block->z_usize);
}

zzip_mem_entry_extra_block(entry, datatype) {
  EXTRA_BLOCK *ext = entry->zz_ext[i];          (vii)
  while (*(short *) (ext->z_datatype)) {
    if (datatype == zzip_extra_block_get_datatype(ext))
      return ext;
    ext += zzip_extra_block_get_datasize(ext);
  }
}
```

(a) Vulnerable code.

```
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000040h   00 00 00 00 00 01 00 00 00 A4 81 00 00 00 00 61
00000050h   55 54 05 00 03 EC E5 71 5A 50 4B 05 06 00 00 00
00000060h   00 01 00 01 00 47 00 00 00 21 00 00 00 00 00
           19 00                                01 00
```

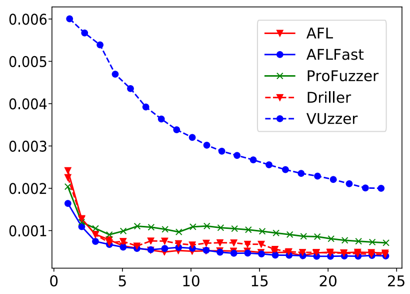
(b) Mutated input.

Fig. 6: Exposing CVE-2017-5975.

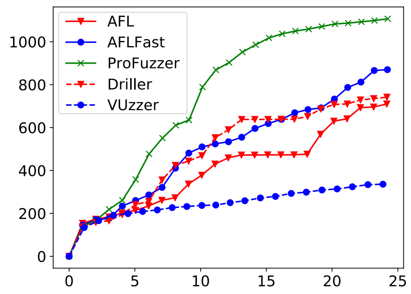
G. Performance

Mutations. As discussed in Section IV, one of the disadvantages of existing fuzzers is that the mutations they perform are largely random due to the lack of input semantics. In the first experiment, we measure the ratio of effective mutations over 24 hours. A mutation is effective when it leads to new coverage. The ratio is the number of effective mutations over the total number of mutations. The results of OpenJPEG are shown in Figure 7a. VUzzer achieves the most effective mutation ratio, since its taint analysis allows VUzzer to produce fewer but more effective inputs than all the other fuzzers that do not use dynamic tainting. ProFuzzer has lower effective mutation ratios at the beginning because of probing, which blindly mutates individual bytes from $0x00$ to $0xff$. After probing, ProFuzzer has the field information to guide mutations, whereas AFL, AFLFast and Driller do not. As a result, their effective mutation ratios drop very quickly, while ProFuzzer can keep a relatively high ratio. In hours 5 and 10, a number of new fields are discovered (by reprobing), leading to the increase of the effective mutation ratio. The experiments on other target programs show similar results.

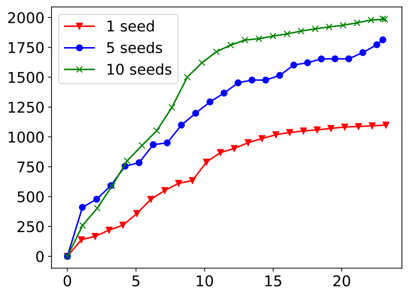
Path Coverage. Figure 7b presents the path coverage in the first 24 hours. In the early stage, the five fuzzers show a similar path coverage growth. After probing, the path coverage of ProFuzzer grows very fast along time. Hour 10 in Figure 7b shows the effect of finding new fields by reprobing. In contrast, AFL mostly performs random mutations. Hence, it sometimes cannot find any new paths for a long time (hours 13-18) until the random walk reaches an interesting space. AFLFast optimizes AFL by prioritizing the seeds that may lead to new coverage through statistical analysis. While it does provide improvement in many cases, it nonetheless lacks field semantics. Hence its improvement is limited compared with ProFuzzer. Driller does not seem to have substantial improvement over AFL as its symbolic execution engine is less effective on real-world applications. VUzzer's performance on path coverage is not as outstanding as its mutation ratio because the heavy-weight taint analysis slows down its execution by about 10 times. Overall, ProFuzzer archives 56%, 27%, 49% and 227% more path coverage than AFL, AFLFast, Driller and VUzzer, respectively, and spends 60%, 54%, 53% and 79% less time to reach the same coverage reported for AFL, AFLFast, Driller and VUzzer in 24 hours. Our evaluation on other target programs shows similar results, see Appendix F.



(a) Effective mutation ratio.



(b) Path coverage.



(c) Sensitiveness to seeds.

Fig. 7: Performance comparison.

Sensitiveness to Seed Inputs. To study how different seed inputs may affect ProFuzzer, we use three different sets of seeds to evaluate path coverage on OpenJPEG. The first one has a single valid seed input, the second and third ones have 5 and 10 valid seed inputs with different combinations of color depth and header size. The result is shown in Figure 7c. As we can see, there are substantial differences between one-seed and five-seed. In contrast, the differences between five-seed and ten-seed are small. Further inspection shows that in one-seed, seven new fields are found within 5 hours, while the remaining three new fields could not be found in 24 hours. With five and ten seeds, these remaining new fields are found within 3 hours. After all the 10 new fields are discovered, the five-seed and ten-seed start to have a similar trend.

VI. Related Work

Field Aware Fuzzing. Closely related to ProFuzzer is field aware fuzzing that tries to understand inputs to help fuzzing. BuzzFuzz [40] is a pioneering work, which locates input bytes that reach attack points and mutates them to the values likely to trigger potential vulnerabilities. Steelix [41] infers magic value bytes from program execution states and replaces them with correct values to pass magic value validation. TIFF [42] performs bug-directed mutations by inferring the program type (e.g., integer, string, etc) of the input bytes. Angora [39] infers the shape and type of input bytes by utilizing the semantics of the instruction that operates on the values (e.g., sign and size of the operands). The identified shape and type information can improve the efficiency of gradient-based search. Other similar works include [10], [43]. ProFuzzer is inspired by field aware fuzzing. Unlike existing techniques, ProFuzzer includes a lightweight mechanism to discover the relations between input bytes and program behaviors by observing the program's execution path variations in response to the mutations of input content, instead of using taint analysis or symbolic execution.

Input Structure Reverse Engineering. Prior approaches use taint analysis or symbolic execution to track the relations of input fields to infer file format [44], protocol [45]–[47] or security configurations [48]. Another line of research infers the program type of a variable (e.g., int and char) by observing its propagation through instructions and system calls (e.g., TIE [25] and REWARDS [24]) or extracts data structures (e.g., struct, array and string) from the memory access patterns during program execution (Howard [26]). These techniques recover types and data structures that are not fuzzing-specific. Consider the variable `header.biBitCount` presented in Code (ii) of Section IV. It will be recognized as an integer by TIE and REWARDS, while it is labeled as an enumeration with four valid values by ProFuzzer.

Hybrid Fuzzing. Recent studies [13], [49] combine fuzzing with concolic execution to address limitations of each individual approach. However, they still face the challenges in scaling to real-world applications due to the performance bottlenecks in their concolic executors. QSYM [14] proposes a practical concolic executor tailored to hybrid fuzzing, which optimizes symbolic emulation and loosens the soundness of constraint solving. Our study shows that ProFuzzer complements QSYM: our exploration and exploitation strategies help to explore more paths and exploit mutation patterns more likely to trigger vulnerabilities. QSYM helps to get through the paths guarded by complex predicate conditions.

Boosting Fuzzing. A number of existing works aim to boost fuzzing by improving seed selection [27], [50], optimizing fuzzing guidance by using statistical metrics [7], [8], [51], [52] or semantics information [53], leveraging fundamental application properties [12] and vulnerability specific patterns [54], reducing path collisions [31], providing new OS level primitives [55], or integrating fuzzing with deep learning [6], [9].

VII. Conclusion

We show that on-the-fly precise input field discovery and semantic understanding can be transparently integrated into regular fuzzing. With the information recovered from probing, field-specific mutations can be performed, leading to improvement of fuzzing performance. Our results show that ProFuzzer substantially outperforms AFL, its optimized version AFLFast, as well as two start-of-art smart fuzzers Driller and VUzzer, and complements a recent work QSYM. It discovers 42 zero-days in two months including 30 new CVEs.

Acknowledgment

The authors would like to thank the anonymous reviewers and Professor Herbert Bos (the PC contact) for their constructive comments. Also, the authors would like to express their thanks to the authors of VUzzer and Angora for the discussion of experiment settings. Purdue authors were supported in part by DARPA FA8650-15-C-7562, NSF 1748764 and 1409668, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. IU authors were supported in part by NSF CNS-1527141, 1618493, 1801432, 1838083, ARO W911NF1610127 and Samsung Gift fund. RUC authors were supported in part by NSFC U1836209 and 61802413. 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] "Oss-fuzz: Five months later, and rewarding projects," https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[2] "How heartbleed could've been found," https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html.

[3] "Peach fuzzer," https://www.peach.tech/products/peach-fuzzer.

[4] "Spike fuzzer," http://resources.infosecinstitute.com/intro-to-fuzzing.

[5] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM Conference on Programming Language Design and Implementation, PLDI 2017*.

[6] 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*.

[7] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*.

[8] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security, CCS 2017*.

[9] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *CoRR*, vol. abs/1711.04596, 2017.

[10] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, SP 2015*.

[11] "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl.

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

[13] 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 the 23rd Annual Network and Distributed System Security Symposium, NDSS 2016*.

[14] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium, SECURITY 2018*.

[15] "Lava-m dataset," http://panda.moyix.net/~moyix/lava_corpus.tar.xz.

[16] "Google fuzzer test suite," https://github.com/google/fuzzer-test-suite.

[17] "Exiv2 issues," https://github.com/Exiv2/exiv2/issues.

[18] "Openjpeg," http://www.openjpeg.org.

[19] "Automatically inferring file syntax with afl-analyze," https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html.

[20] "afl-fuzz: making up grammar with a dictionary in hand," https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

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

[22] "Tcp rfc," https://tools.ietf.org/html/rfc793.

[23] "Wikipedia: Bmp file format," https://en.wikipedia.org/wiki/BMP_file_format.

[24] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010*.

[25] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011*.

[26] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011*.

[27] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium 2014, SECURITY 2014*.

[28] N. Hu and P. Steenkiste, "Quantifying internet end-to-end route similarity," in *Passive and active measurement conference*, vol. 2006.

[29] S. L. Seyler, A. Kumar, M. F. Thorpe, and O. Beckstein, "Path similarity analysis: a method for quantifying macromolecular pathways," *PLoS computational biology*, vol. 11, no. 10, 2015.

[30] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for c++," in *Proceedings of the 24th ACM Conference on Computer and Communications Security, CCS 2017*.

[31] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy, SP 2018*.

[32] "Oss-fuzz," https://github.com/google/oss-fuzz/tree/master/projects.

[33] "The fuzzing project," https://fuzzing-project.org/software.html.

[34] "Syntactically valid files," https://github.com/mathiasbynens/small.

[35] "Afl testcases," https://github.com/mirrorer/afl/tree/master/testcases.

[36] "Libav issues," https://bugzilla.libav.org.

[37] "Mupdf issues," https://bugs.ghostscript.com.

[38] "Zziplib issues," https://github.com/gdraheim/zziplib/issues.

[39] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*.

[40] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009*.

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

[42] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using Input Type Inference To Improve Fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018*.

[43] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *31st IEEE Symposium on Security and Privacy, S&P 2010*.

[44] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz, "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*.

[45] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007*.

[46] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SECURITY 2007*.

[47] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda, "Automatic network protocol analysis," in *NDSS*. The Internet Society, 2008.

[48] R. Wang, X. Wang, K. Zhang, and Z. Li, "Towards automatic reverse engineering of software security configurations," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*.

[49] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering, ICSE 2007*.

[50] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium, SECURITY 2018*.

[51] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2018*.

[52] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 24th ACM Conference on Computer and Communications Security, CCS 2017*.

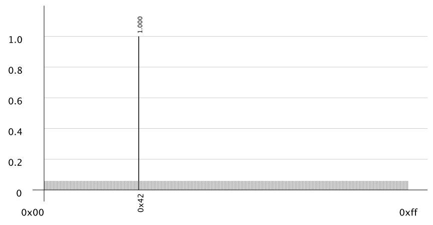
[53] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM Conference on Computer and Communications Security, CCS 2017*.

[54] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations," in *Proceedings of the 22th USENIX Security Symposium, SECURITY 2013*.

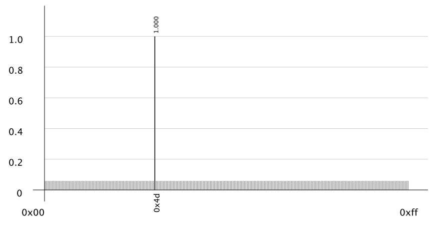
[55] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*.

Appendix

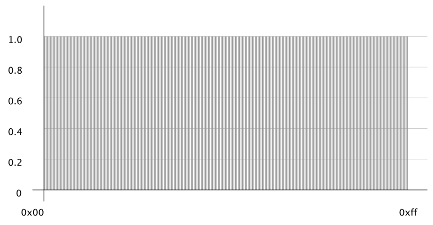
A. Coverage Features



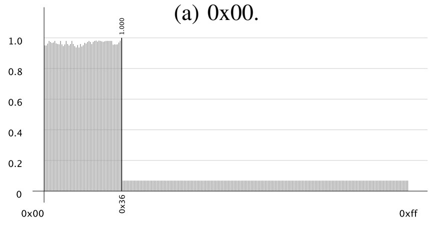
(a) 0x00.



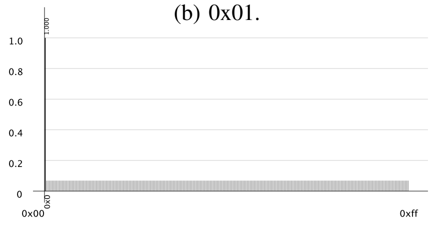
(b) 0x01.



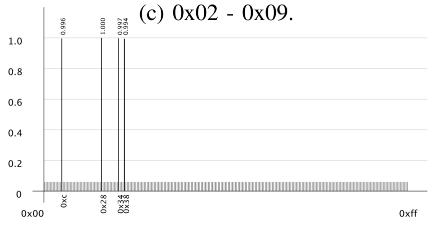
(c) 0x02 - 0x09.



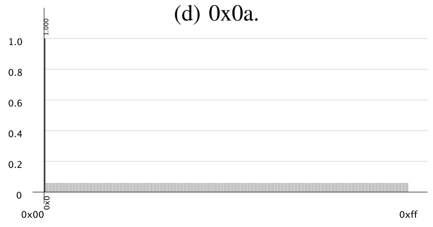
(d) 0x0a.



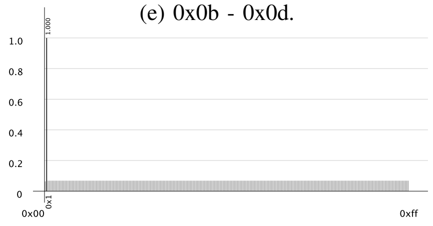
(e) 0x0b - 0x0d.



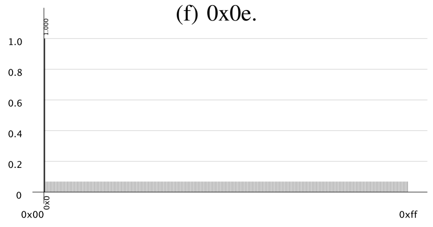
(f) 0x0e.



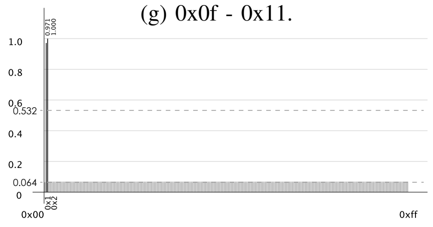
(g) 0x0f - 0x11.



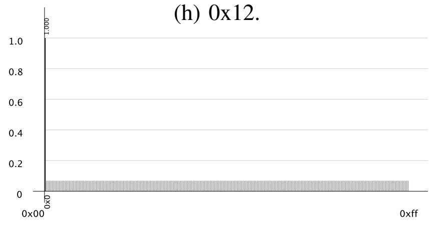
(h) 0x12.



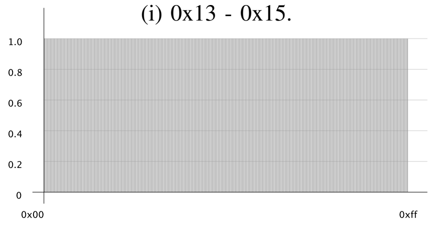
(i) 0x13 - 0x15.



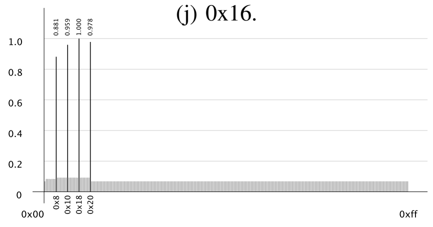
(j) 0x16.



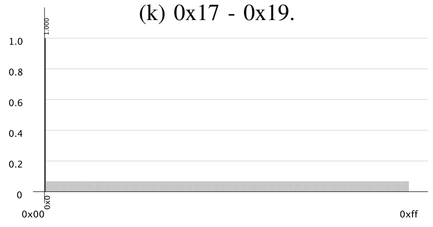
(k) 0x17 - 0x19.



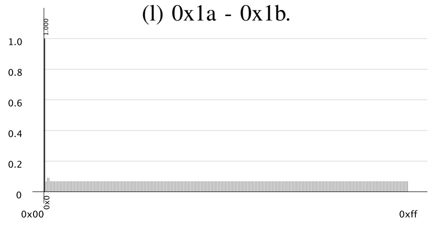
(l) 0x1a - 0x1b.



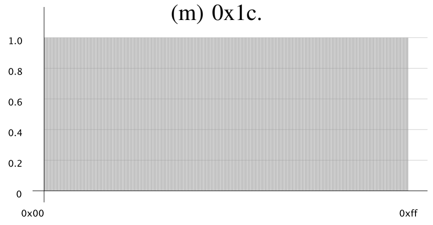
(m) 0x1c.



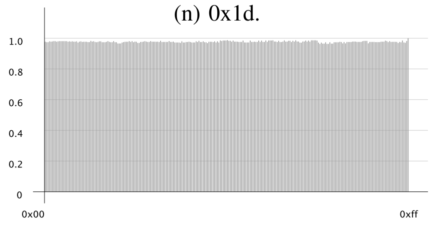
(n) 0x1d.



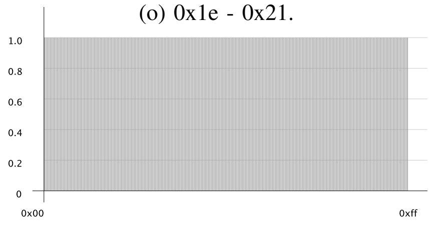
(o) 0x1e - 0x21.



(p) 0x22 - 0x35.



(q) 0x36 - 0x38, 0x3a - 0x3c.

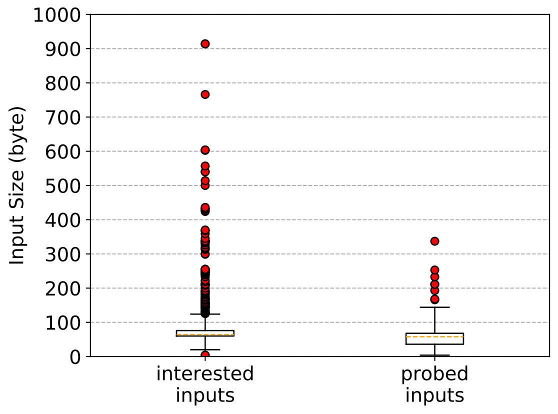


(r) 0x39, 0x3d.

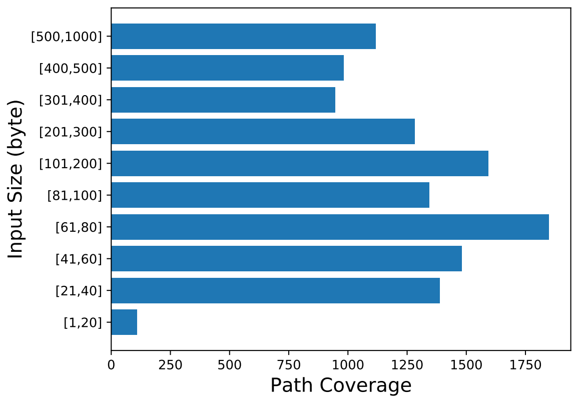
B. Samples of Real-World Programs

Product	Version	SLOC	Deter-ministic	Initial Seeds Size (bytes)				Edge Count		
				Minimum	Maximum	Median	Average	Normal	Exceptional	Similarity
binutils	2.30	388,891	Y	56	256	132	147	256	34	0.111
brotli	1.0.4	35,770	Y	38	1,245	89	184	1,292	51	0.076
elfutils	0.170	115,330	Y	56	256	132	147	629	154	0.187
exiv2	0.26	131,993	Y	108	430	408	376	4,741	1,353	0.287
file	5.33	17,941	Y	5	1,280	10	78	1,762	1,042	0.561
freetype2	2.9.1	2,807	Y	540	1,072	836	809	3,712	1,423	0.212
giflib	5.1.4	8,209	Y	14	1,304	198	507	632	368	0.532
gifsicle	1.91	17,190	Y	14	1,304	198	507	1,635	908	0.398
graphicsmagick	1.3.29	299,186	Y	576	1,024	630	663	3,181	2,511	0.65
guetzli	1.1.0	9,431	Y	107	1,239	285	435	19,584	160	0.012
jasper	2.0.14	41,067	Y	107	1,239	285	435	3,031	343	0.127
jhead	3.0	4,880	Y	107	1,239	285	435	324	47	0.150
json-c	0.13.1	9,009	Y	289	854	501	522	635	53	0.120
json	3.1.2	80,358	Y	289	854	501	522	782	213	0.077
lame	3.100	52,143	Y	462	1,915	879	950	3,434	1,020	0.313
lcms	2.9	43,146	Y	158	3,240	310	1,002	868	117	0.111
libarchive	3.3.2	185,739	Y	10	280	66	91	2,122	1,012	0.360
libav	12.3	703,369	Y	843	2,366	1,493	1,650	7,680	4,459	0.455
libexif	0.6.21	14,032	Y	107	1,239	285	435	224	121	0.347
libjpeg-turbo	1.5.90	63,114	Y	107	1,239	285	435	1,979	133	0.071
libming	0.4.8	72,747	Y	462	1,915	879	950	139	12	0.130
libplist	2.0.0	8,097	Y	56	363	284	251	939	31	0.059
libpng	1.6.34	28,189	Y	67	790	216	264	1,427	256	0.187
libtiff	4.0.9	82,484	Y	108	430	408	376	2,546	107	0.035
libxml2	2.9.7	288,312	Y	7	925	235	329	3,244	1,025	0.367
libyaml	0.1.7	13,529	Y	19	356	34	131	963	123	0.146
lrzip	0.631	19,098	Y	73	1,238	260	575	523	69	0.143
msgpack-c	3.0.1	104,660	Y	240	2,087	420	792	450	39	0.080
mupdf	1.9	102,824	Y	130	758	731	490	14,912	3,013	0.180
nasm	2.13.03	102,523	Y	56	256	132	147	1,074	128	0.288
openjpeg	2.3.0	164,284	Y	576	1,024	630	663	3,543	136	0.050
ots	6.1.1	177,844	Y	540	1,072	836	809	2,864	150	0.037
podofo	0.9.5	78,195	Y	130	758	731	490	3,977	1,062	0.265
tinyxml2	6.2.0	7,560	Y	7	925	235	329	591	52	0.107
udis86	1.7.2	11,083	Y	56	256	132	147	703	208	0.396
vorbis	1.3.6	61,990	Y	462	1,915	879	950	3,974	1,092	0.462
yara	3.7.1	50,983	Y	56	256	132	147	132	37	0.464
zlib	1.2.9	29,675	Y	42	253	70	100	126	80	0.573
zlib-ng	1.0	18,831	Y	41	98	50	62	305	118	0.364
zziplib	0.13.69	12,898	Y	82	162	111	124	166	100	0.350

C. Size Distribution and Path Coverage



(a) Size distribution (openjpeg).



(b) Size and coverage (openjpeg).

D. Zero-day Vulnerability List

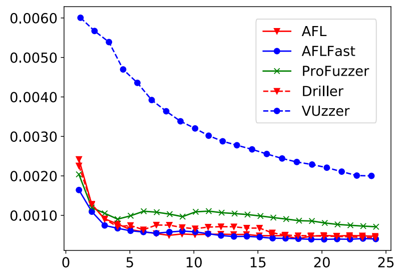
Product	Version	Vulnerability Type	Vulnerable Function	Discovery Date	CVE	Status
exiv2	0.26	heap-buffer-overflow	byteSwap4	2017-12-10	CVE-2017-17723	Fixed
	0.26	heap-buffer-overflow	printStructure	2017-12-10	CVE-2017-17724	Fixed
	0.26	heap-buffer-overread	getULong	2017-12-12	CVE-2017-17725	Fixed
	0.26	uncontrolled-recursion	printIFDStructure	2018-01-17	CVE-2018-5772	Fixed
	0.26	reachable-assertion-failure	readHeader	2017-12-10	CVE-2017-17722	Fixed
	0.26	integer-overflow	floatToRationalCast	2018-01-22		
graphicsmagick	1.3.27	infinite-loop	ReadBMPImage	2018-01-13	CVE-2018-5685	Fixed
	1.3.27	integer-overflow	ReadBMPImage	2018-01-12		
libtiff	4.0.9	uncontrolled-resource-consumption	TIFFSetDirectory	2018-01-18	CVE-2018-5784	Confirmed
	4.0.9	uncontrolled-memory-allocation	cpDecodedStrips	2017-12-05		
	4.0.9	heap-buffer-overflow	extractContigSamples24bits	2017-12-05		
	4.0.9	uncontrolled-memory-allocation	t2p_readwrite_pdf_image	2017-12-05		
	4.0.9	heap-buffer-overflow	PSDataColorContig	2017-12-05		
	4.0.9	uncontrolled-memory-allocation	readSeparateTilesIntoBuffer	2017-12-06		
	4.0.9	uncontrolled-memory-allocation	TIFFReadRawData	2017-12-06		
	4.0.9	uncontrolled-memory-allocation	cpStrips	2017-12-07		
openjpeg	2.3	integer-overflow	opj_j2k_setup_encoder	2018-01-18	CVE-2018-5785	Confirmed
	2.3	integer-overflow	opj_t1_encode_cblks	2018-01-12	CVE-2018-5727	Confirmed
	2.3	excessive-iteration	opj_t1_encode_cblks	2018-02-04	CVE-2018-6616	Confirmed
libav	12.1	invalid-memcpy	ff_mov_read_stsd_entries	2018-01-12	CVE-2018-5684	
	12.1	infinite-loop	event_loop	2018-01-18		
	12.1	invalid-memcpy	av_packet_ref	2018-01-18	CVE-2018-5766	
libming	0.4.8	integer-overflow	readUInt32	2018-01-05	CVE-2018-5294	Fixed
	0.4.8	integer-overflow	readSBits	2018-01-03	CVE-2018-5251	Fixed
mupdf	1.12.0	infinite-loop	pdf_parse_array	2018-01-13	CVE-2018-5686	Fixed
podofo	0.9.5	integer-overflow	ReadObjectsFromStream	2018-01-08	CVE-2018-5309	Confirmed
	0.9.5	integer-overflow	ParseStream	2018-01-06	CVE-2018-5295	Fixed
	0.9.5	uncontrolled-memory-allocation	ReadXRefSubsection	2018-01-06	CVE-2018-5296	
	0.9.5	uncontrolled-memory-allocation	PdfVecObjects::Reserve	2018-01-18	CVE-2018-5783	
	0.9.5	invalid-memcpy	PdfMemoryOutputStream::Write	2018-01-08	CVE-2018-5308	Fixed
	0.9.5	excessive-iteration	PdfParser::ReadObjectsInternal	2018-01-26	CVE-2018-6352	
lrzip	0.631	infinite-loop	unzip_match	2018-01-11	CVE-2018-5650	Confirmed
	0.631	use-after-free	ucompthread	2018-01-17	CVE-2018-5747	Confirmed
	0.631	infinite-loop	get_fileinfo	2018-01-19	CVE-2018-5786	Confirmed
zziplib	0.13.67	memory-alignment-error	__zzip_fetch_disk_trailer	2018-01-31	CVE-2018-5650	Fixed
	0.13.67	memory-alignment-error	zzip_disk_findfirst	2018-02-01	CVE-2018-5641	Fixed
	0.13.67	memory-alignment-error	__zzip_fetch_disk_trailer	2018-01-31	CVE-2018-6484	Fixed
	0.13.67	memory-alignment-error	zzip_disk_findfirst	2018-02-01	CVE-2018-6542	Fixed
	0.13.67	infinite-loop	unzzip_cat_file	2018-01-26		Fixed
	0.13.67	out-of-bounds read	main	2018-01-26		Fixed
	0.13.67	out-of-bounds read	zzip_disk_fread	2018-01-28	CVE-2018-6381	Fixed
	0.13.68	uncontrolled-memory-allocation	__zzip_parse_root_directory	2018-02-08	CVE-2018-6869	Fixed

E. Bugs Found in LAVA-M

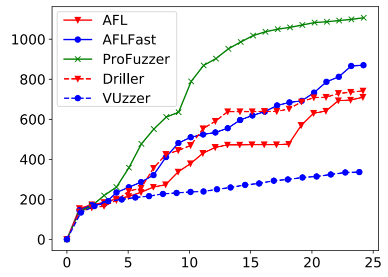
Original ProFuzzer	
Program	**Bugs ID**
base64	1, 222, 235, 284, 386, 774, 778, 786, 804, 805, 806
md5sum	270, 353, 380, 549
uniq	112, 166, 215, 222, 318, 321, 322, 346, 347, 393, 396, 443, 447
who	1, 3, 7, 8, 14, 18, 20, 26, 79, 83, 138, 319, 474, 587, 1816, 3800, 4358

Optimized ProFuzzer	
Program	**Bugs ID**
base64	1, 222, 235, 253, 276, 278, 284, 386, 554, 556, 560, 576, 583, 584, 774, 786, 788, 805, 806, 813, 815, 817, 841, 843
md5sum	1, 268, 269, 270, 335, 341, 353, 380, 549, 571
uniq	112, 130, 166, 169, 170, 171, 215, 222, 293, 296, 297, 318, 321, 322, 346, 347, 368, 371, 372, 393, 396, 397, 443, 446, 447, 468, 471, 472
who	1, 3, 4, 5, 7, 8, 9, 10, 14, 18, 20, 22, 26, 55, 57, 59, 60, 61, 62, 75, 79, 83, 87, 89, 137, 138, 150, 159, 319, 341, 474, 475, 587, 985, 1804, 1816, 3800, 3967, 4358, 4362, 4364

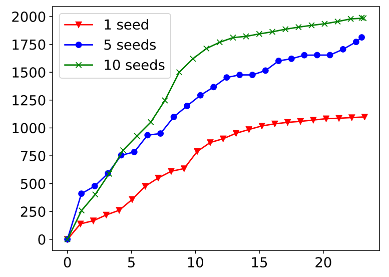
F. Performance Details



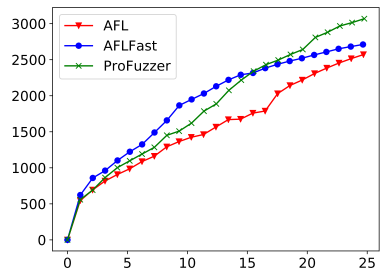
(a) Effective mutation ratio (openjpeg).



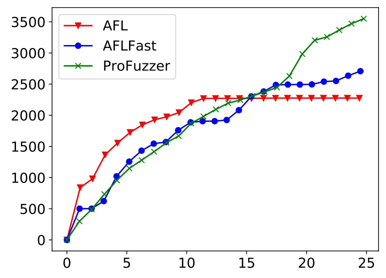
(b) Path coverage (openjpeg).



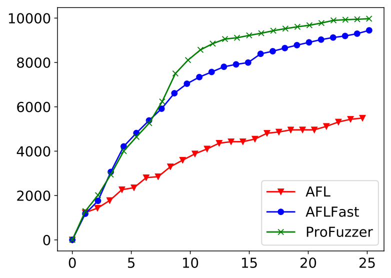
(c) Sensitiveness to seeds (openjpeg).



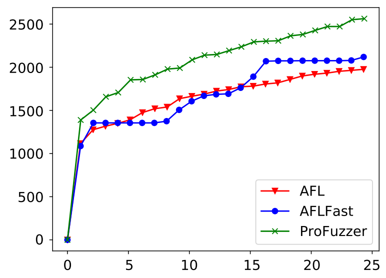
(d) Path coverage (exiv2).



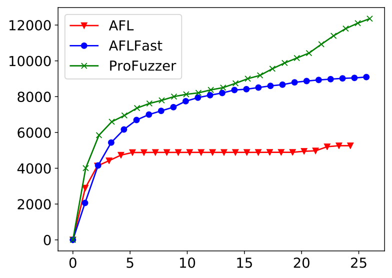
(e) Path coverage (graphicsmagick).



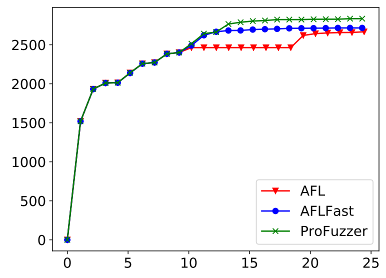
(f) Path coverage (libtiff).



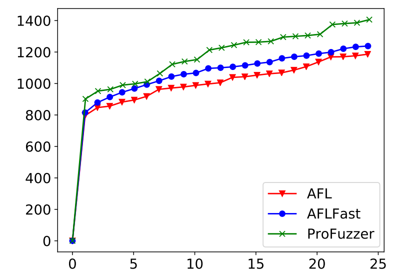
(g) Path coverage (libav).



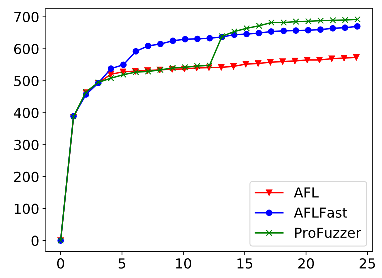
(h) Path coverage (libming).



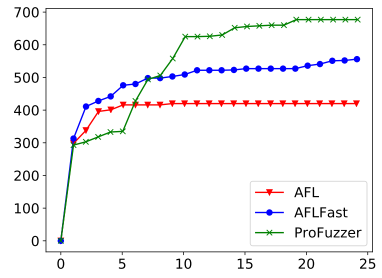
(i) Path coverage (mupdf).



(j) Path coverage (podofo).



(k) Path coverage (lrzip).



(l) Path coverage (zziplib).