Field-aware Evolutionary Fuzzing Based on Input Specifications and Vulnerability Metrics

Yunchao Wang, Zehui Wu, Qiang Wei and Qingxian Wang
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan Province, China
Corresponding author: Qiang Wei (e-mail: prof_weiqiang@163.com).

Abstract—**Evolutionary fuzzing technology based on genetic algorithm has become one of the most effective vulnerability discovery techniques due to its fast and scalable advantages. How to effectively mutate the seed input plays a crucial role in improving the efficiency of the fuzzing. A good mutation strategy can increase code coverage and vulnerability triggering probability. Existing fuzzing tools generally focus on how to mutate smartly to improve code coverage to find more vulnerabilities (such as passing the branch with magic bytes), but they still face two challenges which substantially reduces the efficiency of vulnerability discovery. First, the input space is huge and current fuzzers are not aware of the input format, resulting in many mutated inputs are invalid. Second, they believe all bytes are equal and mutate them sequentially, wasting lots of time testing some uninteresting bytes. To this end, this paper proposes a field-aware mutation strategy that can find more vulnerabilities by generating fewer but more effective inputs. Specifically, we extract the field and type information of the seed input through the existing input specifications to ensure that the mutation is performed in field level instead of byte level and the optimal mutation strategy is selected. At the same time, the input fields are scored by code assessment based on vulnerability metrics, thus the more important fields (i.e., fields that are more likely to trigger the vulnerability) are prioritized to be mutated. We implemented a prototype tool, FaFuzzer, and evaluated it on two different datasets consisting of a variety of real-world applications. Experiments show that our field-aware strategy can find more vulnerabilities with fewer inputs than existing tools, while maintaining high code coverage. We found many unknown bugs in five widely used real-world applications and reported them to the relevant vendors.**

Keywords-evolutionary fuzzing; vulnerability discovery; input specifications; vulnerability metrics; software testing

I. INTRODUCTION

With the outbreak of many attack events such as Heartbleed attack [1], WanaCry ransomware [2], people's property and privacy have been greatly threatened. The reason behind them is that vulnerabilities existed in the related software can lead to control flow hijacking or information leakage. Therefore, eliminating software vulnerabilities becomes critical. Fuzzing is currently the most popular and effective technology for discovering software vulnerabilities and bugs. It has been used to discover a large number of high-impact security vulnerabilities, such as CVE-2014-0160 [3], CVE-2017-7668 [4] and so on. Its idea is to generate some semi-valid input to trigger some anomalous behavior in software by mutating a set of initial inputs according to some specified strategies. In terms of ways of sample generation, it is divided into generation-based fuzzing and mutation-based fuzzing. The generation-based fuzzer (eg. Peach [5]) relies on the syntax for describing the input structure to generate test cases. Its advantage is that it can generate correctly formatted samples, to avoid being discarded early in the parsing stage of the program, which can guarantee to trigger some deep code. But it needs to manually write a syntax file such as Peach Pit, which is time consuming and error-prone, and it has no coverage feedback mechanism, which also limits its ability to explore code. Mutation-based fuzzing (eg. AFL [6]) does not rely on input syntax and randomly mutate the initial seed inputs to generate new test cases. This method is convenient and easy to use, but because it is lack of the input structure awareness, resulting in generating a lot of invalid input and wasting a lot of time.

Evolutionary fuzzing is a mutation-based technology and adopts the idea of genetic algorithms. It usually contains a fitness function to measure the current seed based on the execution result, thus ensuring that only better seeds are retained during the iteration. Obviously, this idea can effectively improve code coverage compared to the traditional black box testing tools (e.g., zzuf [7]). The mutation operator is a key component of one fuzzer. The input generated during the previous iterative process is mutated to obtain new test cases according to different mutating strategies. AFL is a typical evolutionary fuzzer and has made great success. However, as we all know it has a great limitation of not understanding the input structure and seeing the input as sequence of consecutive binary bytes. The mutation strategy it adoptes is completely random and can lead to a lot of uninteresting or useless input, therefore many prior studies [8, 9, 10] have tried a variety of intelligent mutating strategies. Diller [8] and DigFuzz [9] uses symbolic execution to solve complex condition constraints which is very hard for fuzzer to satisfy. Mohit Rajpal et al. [10] present a learning technique that uses neural networks to learn promising bytes from past fuzzing exploration to guide future mutation. VUzzer [11] is a recent research which leverages dynamic taint analysis (DTA) to identify byte offsets used in compare instruction or function and mutates them with inferred interesting values. We can find that these strategies are all proposed from the perspective of how to improve code coverage, rather than maximizing probability of triggering vulnerabilities. On one hand, they only focus on how to mutate

This work was supported by National Key R&D Program of China under Grant 2017YFB0803202.

978-1-7281-0945-9/19/$31.00©2019 IEEE

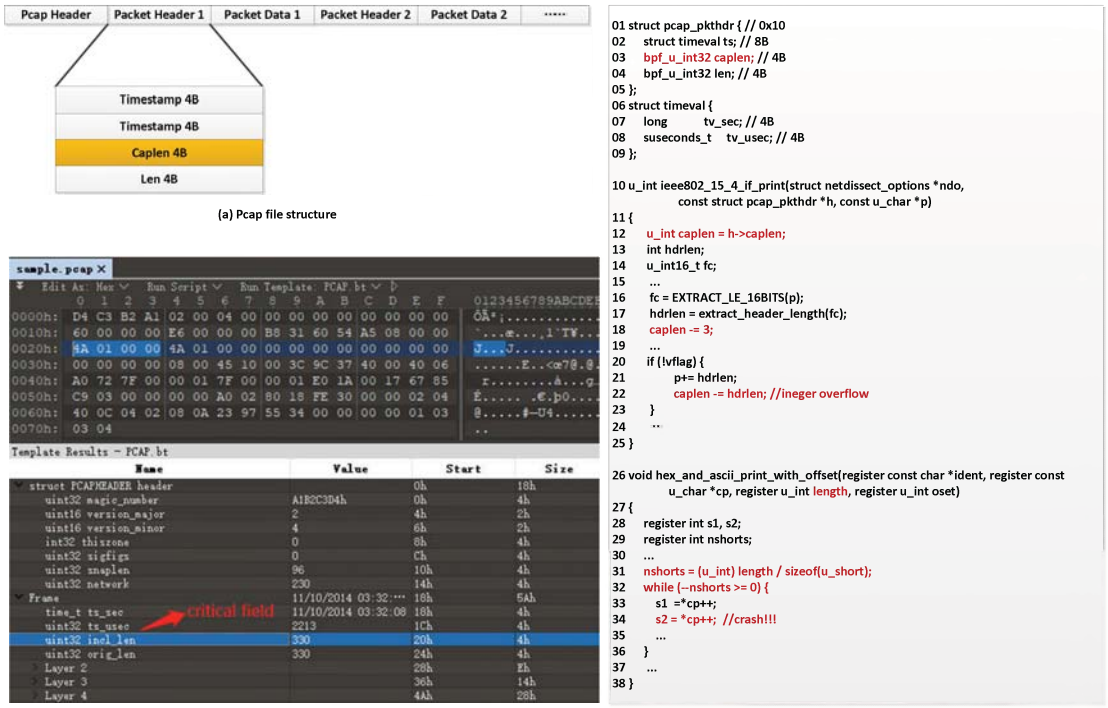

Figure 1. Motivating example

bytes which affect branch jump direction (e.g., Magic bytes), but ignore the effect of other bytes. Of course, it could help break through some complex condition branch to some extent, but they are still not aware of most of the input offsets and types and resort to mutate blindly and randomly, resulting in many of the mutated inputs are invalid or useless. On the other hand, they treat all bytes equally and mutate them sequentially, so that some unimportant bytes (irrelevant to vulnerability) are mutated which wastes lots of time. Therefore, current mutation strategies limit the effectiveness and efficiency of discovering vulnerabilities from the above two aspects.

In fact, the input space is very large, for example, for one input of 1kb size, it may generate inputs. We can't completely traverse it within a limited amount of time, so we can only choose more efficient and meaningful inputs. This paper focuses on how to perform more effective mutations in seed inputs to increase the probability of vulnerability discovery. We notice that the effectiveness of a fuzzer relies on its understanding of inputs which are often structured data and can be partitioned into a sequence of data fields with specific semantics. Therefore, we propose a field-aware mutation strategy in this paper. First, we use existing input specification to extract field and type information, which help mutates in field level and chooses optimal value to use for the mutation. Then, we leverage code assessment method based on vulnerability metrics to determine which fields are more important and should be prioritized to be mutated to enhance the chance of hitting zero-day targets.

We implemented our prototype tool called FaFuzzer based on VUzzer and evaluated on the datasets with 18 known vulnerabilities and 5 real-world applications with unknown vulnerabilities. We found that FaFuzzer can find more vulnerabilities with less inputs within the 12-hour time limit and code coverage increased significantly compared with VUzzer and AFLPIN [12], a pintool implementation of AFL based on PIN [13]. In summary, the main contributions of this paper are as follows:

- A field-aware mutation strategy based on known input specification and vulnerability metrics is proposed. The mutation is performed at the field level and important fields are prioritized, which can effectively reduce input search space and expose vulnerabilities faster.
- We implement one prototype tool, called FaFuzzer, and evaluate it on two different datasets. The experiment indicates the effectiveness of our method in terms of vulnerability detection capability as well as code coverage.
- We find some unknown bugs in 5 real-world applications and reported them to the relevant vendors which helps improve the applications' security.

II. MOTIVATION

We illustrate the problems of traditional fuzzing method and the key idea of our approach through a real vulnerable example [14]. We consider a common input format (pcap) and its processing program (tcpdump).

One pcap file usually includes the pcap file header, followed by one or more packet headers and packet data, as shown in Fig. 1(a). The file header occupies 24 bytes, the next 16 bytes are the packet header, which contains 4 fields, each field occupies 4 bytes, and the third field is the *caplen* field, meaning the length of currently captured packet data. After this length is the location of the next package. Fig. 1(b)

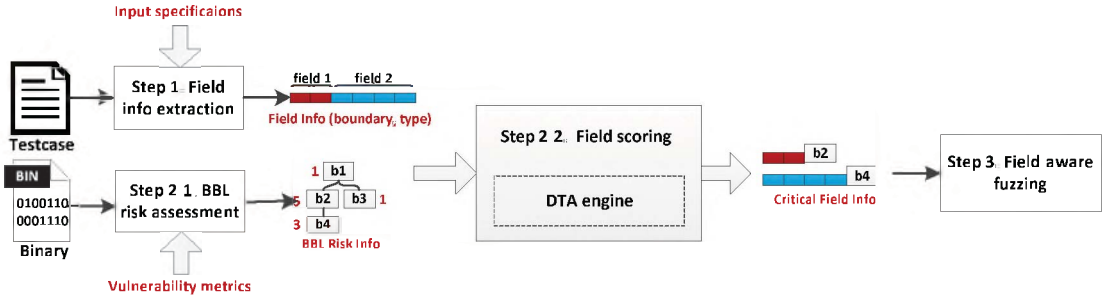

Figure 2. Architecture of FaFuzzer

is a pcap sample file, we can clearly see the field offset, size and type information. Fig. 1(c) shows a vulnerable code snippet that exists in tcpdump version 4.5.1. Crafted *caplen* field in the pcap file may cause integer overflow and the overflowed value is used as loop counter, resulting in invalid memory access and program crash. Specifically, in the function *ieee802_15_4_if_print*, the *caplen* field (uint32 type) is parsed in line 12. Unfortunately, the program does not perform any sanity check. If the field is changed to a value less than 0x15, it will become a negative value after two subtraction operations in line18 and line22 respectively. Since the field is a uint32 type, it will be overflowed and become a large positive value. This value is finally passed to function *hex_and_ascii_print_with_offset* via function call and is then used in the loop counter of the while loop in line 33, so that pointer deference operation in line 34 access invalid memory and program crash in line 35. We tested this program with VUzzer and AFL. We found that it took 9665 and 65832 inputs respectively for VUzzer and AFL to crash the application, but FaFuzzer triggered the crash with only 650 inputs.

Through our analysis, the reasons behind this include the following two aspects:

1. **Current fuzzers are unaware of the input structure.** They treat the input as a sequence of consecutive binary bytes and have no idea of the type of byte offsets. They perform random bit flips during the process of mutation. This method is completely by luck, in other words, they still spend a huge amount of mutation time and effort on triggering a bug by constantly trying. Although the speed of fuzzing is very fast, but actually a large number of generated inputs are invalid or uninteresting, resulting in being rejected in the early syntax parsing stage of target program. In the above example, VUzzer and AFL are not clear that the bytes from offset 0x20-0x23 is a separate field and its type is uint32. It randomly selects consecutive subsequences and perform random bit flip, the search space is hence massive, but FaFuzzer can target the field and mutate with intended offset value according to its type (e.g., 0, maximum, etc.), which can trigger the crash faster.

2. **Current fuzzers do not distinguish the importance of different bytes.** They treat all bytes in the input file equally. Previous research [15] has pointed out that mutating all bytes blindly, the efficiency is very low. In fact, not all bytes are meaningful for vulnerability discovery. Mutating some bytes (e.g., operands of compare instructions or functions) is likely to explore new execution paths and mutating some other bytes (e.g., operands of dangerous instructions or functions) is more likely to trigger a vulnerability, which both depend on how these bytes are used by the program. They ignore the fact that some input bytes may contribute more to vulnerability discovery. In the above example, VUzzer and AFL mutate the bytes in the input from the beginning to the end. However, FaFuzzer can determine the *caplen* as an important field by means of code assessment because it is used in a lot of dangerous operations, for example, performs arithmetic operations and is used in loop counter. Therefore, when testing with FaFuzzer, this field will be prioritized to mutate.

Therefore, from the above example we can learn that mutating with field information and prioritizing important fields can improve the both the effectiveness and efficiency of vulnerability discovery.

III. METHODOLOGY

To address the aforementioned limitations, we propose our solution, FaFuzzer, and its architecture is shown in Fig. 2. Firstly, when a test case is selected for mutation, the field (start and end index) and type information (char, UInt, etc.) are automatically extracted according to the input specifications. Secondly, all basic blocks in the binary program are assessed according to the vulnerability metrics and return the metric value of each basic block. The vulnerability metrics here are features extracted through analysis on a large number of vulnerable code, such as dangerous instruction or API. Thirdly, DTA is used to map the field and the tainted basic block, and finally each field can be scored according to the metric value of each basic block in the execution trace affected by this field. Finally, the higher scored fields are believed of critical importance (easier to trigger the vulnerability) and should be preferentially mutated and the type information is also considered for optimal mutation strategy. Our field-aware fuzzing is based on VUzzer's evolutionary fuzzing loop, and we modify its mutation operator to prioritize critical field and perform type-based mutation. In the following sections, we elaborate each step of the proposed approach.

A. Field Information Extraction

As we all know, inputs are usually composed of a series of fields with specific semantics. It is the task of the program to make sense of these fields, i.e., to parse the file and process the relevant information. So, if the input is invalid, it is more likely to be rejected early in the parse stage. From the previous example, we can see that field information is very important for guiding fuzzing. First, the field offset information can be used to achieve field-level mutation which can help reduce the search space and generate more valid input. Second,

leveraging field type information the fuzzer can operate in a more intelligent way, focusing only on the subspace of inputs most likely leading to vulnerabilities by using some interesting values. So before mutating the selected seed input, we first need to extract the field information in the input automatically to achieve more effective mutation (Step 1) as shown in Fig. 2. In the previous study [16, 17], the reverse engineering of the input format based on dynamic taint analysis was proposed, but this method suffers from the problems of inefficiency and inaccuracy. According to our observations, most of many popular input formats (e.g., png, bmp, mp3, mp4, elf, zip, etc.) have known input specifications that accurately describe the format information of the file. For example, 010editor [18] is a tool that can view binary file formats and integrates many template files [19] and these template files can be used to parse the file structure and obtain the field information. Fortunately, the existed python script tool PFP [20] can parse the target file according to the 010Editor's template and parse the input stream of a file into the corresponding data structure. We call the PFP's interface pfp.parse with the sample file and the template file as input arguments as shown in Fig. 3(a), then it will return a DOM object which we can use to obtain all field information as shown in Fig. 3(b). We can see that offset (as marked in red) and type information (as marked in blue) of each field can be recovered, and the field type covers common data type in C/C++ programming language, such as UShort, UInt, char, char*, Enum and so on.

```
dom = pfp.parse(
        data_file="~/sample.png",
        template_file="~/PNG12Template.bt"
)
print(dom._pfp__show(include_offset=True))
```

(a) PFP interface

```
0000 struct {
  0000 myPngFile = 0000 struct {
    0000 sig      = 0000 struct {
      0000 btPngSignature = UShort[4]
    }
    0008 ihdrChunkHdr = 0008 struct {
      0008 btChunkLen = UInt(13 [0000000d])
      000c btChunkType = Char[4] ('IHDR')
    }
    0010 ihdrChunkData = 0010 struct {
      0010 width    = UInt(20 [00000014])
      0014 height   = UInt(20 [00000014])
      0018 bit_depth  = Char(8 [08])
      0019 color_type = Enum<Char>(6 [06])(AlphaTrueColor)
      001a compr_method = Enum<Char>(0 [00])(Deflate)
      001b filter_method = Enum<Char>(0 [00])(AdaptiveFiltering)
      001c interlace_method = Enum<Char>(0 [00])(NoInterlace)
    }
    001d ihdrCrc   = UInt(2374573325 [8d891d0d])
    /* some other chunks*/
    00ac chunk_iend = 00ac struct {
      00ac iendCrc   = UInt(2923585666 [ae426082])
    }
  }
}
```

(b) field info extracted by PFP

Figure 3. Example of PFP on png file

B. *Critical Field Identification*

After the first step, we can understand all the field information in the input, but as mentioned in section 2, one input is usually composed of many fields and mutating them blindly is very time consuming. Identifying critical fields in the input can help focus on testing input offsets that easily trigger vulnerabilities within limited computational and time resources, thus improving the efficiency of vulnerability discovery. The importance of one field relies on how this field are used in the program. Intuitively, if a field is used in some risky operations or code area, then this field is important, that is, mutating the important field will increase the probability of triggering the vulnerability in code area affected by this field. We achieve critical field identification in two steps (Step 2-1 and Step 2-2) as shown in Fig. 2. First, we need to conduct risk assessment for the basic blocks in the program to determine the risk value of each basic block. Second, the importance of fields is scored according to the risk value of each basic block and mapping relationship between tainted basic blocks and source fields.

TABLE I. VULNERABILITY METRICS

Dimension	Example	Metric Description
Function	strcpy, memcpy, sprintf, malloc, free, etc.	Number of dangerous library functions in bbl
Instruction	rep movsd, mov [reg], reg, mul, sub, add, etc.	Number of dangerous instructions in bbl
Location	for {bbl}, etc.	Number of nested loops of bbl

Note that in this paper, we choose basic blocks as target rather than functions, because some functions in the program are particularly large, but dangerous code only cover few basic blocks. One function may access a lot of fields, but not all fields are used in dangerous code area. For this reason, if we do function risk assessment, the granularity is too coarse, resulting in inaccurate metric results. We consider the basic block risk assessment from three dimension as shown in Table I.

1) Dangerous library functions. Many vulnerabilities are caused by some unsafe memory or string manipulation functions. Therefore, if there is such function call without sanity check in a basic block, it is more likely to cause buffer overflow or other memory corruption vulnerabilities.

2) Dangerous instructions. Similar to calls of dangerous library functions, some specific instructions can also lead to vulnerabilities. For example, loop-memory-write instructions (such as reps movsd) may cause buffer overflow, arithmetic instructions may cause integer overflow or float point exception, and memory read and write instructions may cause illegal memory access.

3) Location of the basic block. It is well known memory read and write operations in the loop often lead to information leakage or buffer overflow. So, if one basic block is located in one or more loops, then it is deserved to be tested. The general idea behind this is that code areas with a high complexity may be of low quality and difficult to test and maintenance so that more bugs may exist.

It should be noted that the basic block risk assessment is one-time effort and completed by static analysis before fuzzing, so it will not reduce the efficiency. We use IDAPython [21] script to obtain the three types of risk values. Given these three metric values for basic blocks in the target application, we compute a risk value (denoted as RV) for each basic block by adding up all the metric values. Formally, for a program P that has γ basic blocks, it can be denoted as $P = \{b_1, b_2, \ldots b_\gamma\}$, where b_i represents the i-th $(1 \le i \le \gamma)$ basic block. The number of dangerous library functions it contains is denoted as $N_{func}(b_i)$, the number of dangerous instructions is denoted as $N_{ins}(b_i)$, and the number of loop nesting layers in which the

current basic block is located denoted as $N_{loop}(b_i)$. Then the risk value of the i-th basic block is represented as (1).

$$RV(b_i) = N_{func}(b_i) + N_{ins}(b_i) + N_{loop}(b_i) \quad (1)$$

After a test case is executed, an execution trace composed of basic blocks is generated. And DTA can be used to monitor the flow of tainted input and trace back tainted values to individual field in the input, that is, which basic blocks are tainted by which fields. Finally, the score of the corresponding fields can be calculated by the risk value of the tainted basic blocks. Formally, for one input I that has λ fields, it is denoted as $I = \{f_1, f_2, \ldots f_\lambda\}$, where f_i represents the i-th $(1 \le i \le \lambda)$ field. Then the score of field f_i can be represented as (2), where $taintSrc(b_m)$ is the set of all fields that taint basic block b_m.

$$Score(f_i) = \sum_{b_m \in P \wedge taintSrc(b_m) \neq \varnothing} RV(b_m) \quad (2)$$

If $Score(f_i)$ is greater than a certain threshold, then this field is critical. We prioritize all critical fields and other fields will not be tested, thereby saving lots of testing time. For example, as shown in Fig. 4, the dynamic trace obtained by executing program P with I as input is indicated by a red line, that is, *b1->b2->b4->b7*. We can determine source fields of each tainted basic block by DTA, that is, $taintSrc(b_1)=\{f_1\}$, $taintSrc(b_2)=\{f_2\}$, $taintSrc(b_4)=\{f_2\}$, $taintSrc(b_7)=\{f_3, f_4\}$, then we can calculate score of each field, $Score(f_1)=RV(b_1)=2$, $Socre(f_2)=RV(b_2)+RV(b_4)=7$, $Score(f_3)=Score(f_4)=RV(b_7)=2$. Therefore, the field f_2 has the highest score and should be mutated first which will increase the probability of triggering vulnerability in basic block b_2 and b_3.

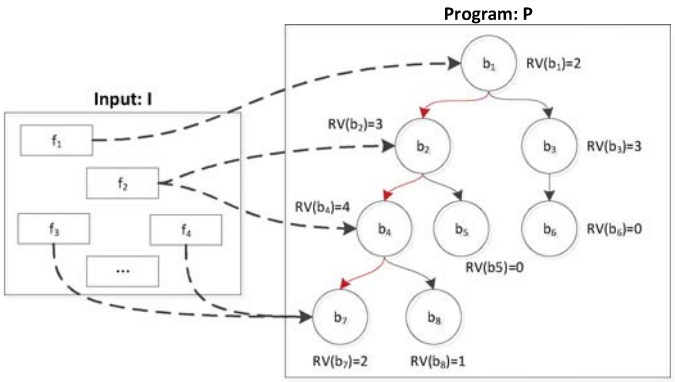

Figure 4. Taint flow between input fields and bbls

C. *Field-aware Fuzzing*

After the first two steps, given one testcase, we can automatically recover and understand the field and type information, as well as fields that are critical to vulnerability discovery, which can be used to guide smart mutation during fuzzing to increase the chances of finding a 0day vulnerability. First, we can perform mutation in field level with field offset information to improve the validity of generated input. For example, if encountering one field consisting of 1 byte, VUzzer may mutate together with surrounding bytes which increase the probability of generating invalid input because it is unaware of the input structure. However, FaFuzzer would mutate this single byte. Second, the field type is also important for meaningful mutation. For example, if encountering one field is of type UINT32, VUzzer will waste a considerable amount of its mutation effort on mutating at such offsets with arbitrary values. FaFuzzer would mutate these 4 bytes together instead of mutating only one single byte and at the time prioritize interesting boundary values (e.g., 0 or 65535 etc.) to modify them expecting to trigger integer overflow. If one field is of type char*, we can increase the size of strings by inserting byte strings of some arbitrarily chosen length. The place where we add the additional sequence of bytes is chosen randomly between the starting offset and ending offset of the string. At last, when we know critical fields in the input, FaFuzzer can prioritize them to mutate, which can expose vulnerability faster and reduce the input search space.

Our field-aware fuzzing approach is based on the VUzzer's framework and is shown in algorithm 1. First, we assess all the basic blocks in the program by static analysis before fuzzing and get risk value of each basic block (line 1). Second, we add the field information extraction before the mutation of the selected seed input (line 4). Then we add the field scoring method after DTA module (line 5, 6). Finally, VUzzer's mutation strategy is modified (line 7) based on the score of each field. Through our field-aware fuzzing, we can prioritize critical fields as well as optimal mutation strategy at these offsets to improve the vulnerability trigger probability. It should be noted that fields used in the compare instructions or functions are still mutated following VUuzzer's original mutation strategy regardless of their priority, because mutating these fields plays a critical role in improving code coverage.

Algorithm 1 Field-aware fuzzing

Input: Seeds, Target program P, Vulnerability metrics M, Input specifications S
Output: Crash inputs
1: rv[bbl, riskvalue] = BBLAssessment(P, M)
2: **repeat**
3: **for** seed in Seeds **do**
4: offset, type = ExtractFieldInfo(seed, S)
5: taint[offset, bbl]= DTA(P, seed)
6: score[field, scorevalue]=FieldScore(taint, rv)
7: newinput = Mutate(seed, offset, type, score)
8: result, cov=Execute(newinput)
9: **If** result is crash **then**
10: Append newinput to Crash inputs
11: **End if**
12: **If** HasNewCov(cov) **then**
13: Append newinput to seeds
14: **End if**
15: **End for**
16: **until** TERMINATION CONDITION is met

IV. EVALUATION

In order to measure the effectiveness of our proposed fuzzing technique, this section presents an evaluation of FaFuzzer. We evaluate FaFuzzer on two different datasets, the real-world applications with known vulnerabilities and the latest version with unknown vulnerabilities respectively. And these applications can handle multiple types of input format, such as image, audio, video, compression and so on. In addition, we compare with state-of-the-art fuzzers such as VUzzer and AFLPIN to show how effective FaFuzzer is in

TABLE II. RESULTS OF KNOWN VULNERABILITIES EXPOSURE

Input Format	Application	Vulnerability	FaFuzzer(#inputs)	VUzzer(#inputs)	AFLPIN(#inputs)
TIFF	libtiff	CVE-2018-12900	8.2K	28.5K	-
		CVE-2018-17000	5.4K	-	-
PNG	libpng	CVE-2018-13785	8K	-	-
		CVE-2018-14048	-	-	-
BMP	openjpeg	CVE-2018-5727	3.4K	15.2K	48K
		CVE-2018-5785	10.8K	30.5K	55K
		CVE-2018-6616	-	-	-
ZIP	zziplib	CVE-2018-7725	11.5K	-	-
		CVE-2018-7727	22K	-	-
ELF	binutils	CVE-2018-18700	1.2K	9.8K	-
		CVE-2018-18701	2.9K	17.6K	21.5K
		CVE-2018-20002	-	-	-
SWF	libming	CVE-2018-8806	0.8K	8.5K	24.3K
		CVE-2018-8961	1.5K	-	-
WAV	audiofile	CVE-2017-6827	3K	4.8K	16.5K
		CVE-2017-6828	5.1K	16.2K	-
AVI	libav	CVE-2018-5766	9.5K	22.3K	-
		CVE-2018-5684	10.3K	-	-
Total		**18**	**15**	**9**	**5**

identifying bugs with much fewer but meaningful inputs. We ran all our experiments on an Ubuntu 14.04 system equipped with a 2-core Intel CPU and 16 GB RAM. All programs are fuzzed for maximum 12 hours to show the importance of generating high-quality inputs in less time. It's important to note that we're emphasizing how to quickly generate more valid inputs instead of execution speed. Therefore, we choose to compare against AFLPIN instead of AFL. FaFuzzer, VUzzer and AFLPIN are all implemented with the dynamic instrumentation tool PIN and runs at an order of magnitude, while the AFL's fork mode guarantees faster execution speeds so that more inputs can be generated per unit of time, despite that most of mutated inputs are of low-quality.

A. *Known Vulnerability Exposure*

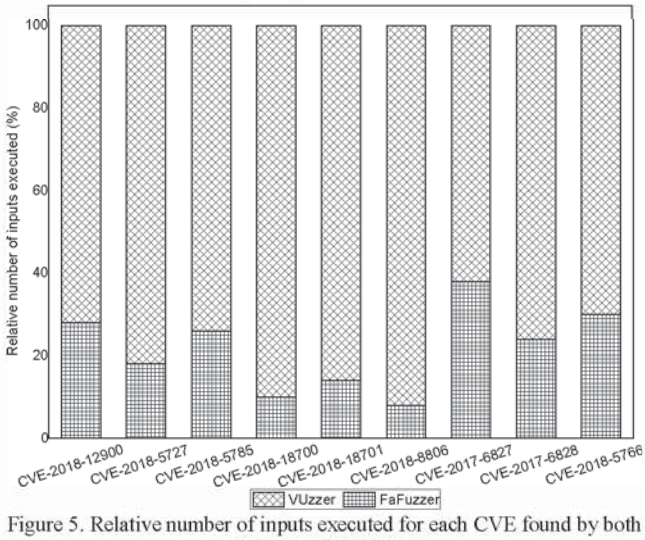

Figure 5. Relative number of inputs executed for each CVE found by both VUzzer and FaFuzzer.

We choose 8 popular programs that process a variety of input format as shown in Table II. All these input format have known and public input specifications, thus our approach can be well applied. We manually collect 18 vulnerabilities from the CVE vulnerability database [22], then download the source code of vulnerable version and compile into binary programs. We run the three fuzzers on them to see whether they can expose all these vulnerabilities and how many inputs are generated to expose the known CVE. In addition, for each application, we gather 10 unequal and valid inputs as seed inputs for each fuzzer considered in our evaluation. From Table II, we can see that FaFuzzer found 15 out of 20 known vulnerabilities, while VUzzer and AFLPIN discovered 9 and 5 respectively. We use "-" to indicate that the vulnerability is not found by corresponding fuzzer. The vulnerabilities discovered by the other two fuzzers are all covered by FaFuzzer with much less mutated inputs at the time. AFLPIN performed worst due to its complete lack of input semantics. And Fig. 5 depicts the relative number of executions for the cases where both VUzzer and FaFuzzer found CVEs (9 in total), further evidencing that FaFuzzer can significantly prune the search space compared to VUzzer.

B. *Unknown Vulnerability Discovery*

In this section, we perform testing on 5 real-world applications which are commonly used in a large number of third-party applications or frameworks and have been tested many times in previous studies. They are all in latest version when tested. The exiv2 and libav can process a variety of image and audio/video formats respectively. As shown in Table III, FaFuzzer found more crashes and covered more basic blocks with fewer inputs compared to VUzzer and AFLPIN. This confirms our observation that field-aware mutation strategy can improve the efficiency of vulnerability discovery as well as code coverage. Finally, we further confirm 7 unknown bugs by manual analysis of these crashes. Among them, we found one heap overflow in libtiff library which processes images of TIFF format, neither VUzzer nor AFLPIN found it. We reported the problem to the vendor, and they confirmed and fixed it [23]. Fig. 6(a) shows the distribution of unique crashes found in exiv2 over a period of 12 hours. We can see that not only can FaFuzzer find more bugs, but also find them sooner. This is mainly because FaFuzzer knows which bytes are more important and likely to trigger vulnerabilities. Fig. 6(b) depicts the effective mutation ratio achieved by the three fuzzers. The ratio is the number of effective mutations over the total number of mutations and we believe a mutation is effective when it leads to new coverage.

TABLE III. RESULTS OF UNKNOWN VULNERABILITIES DISCOVERY

Input format	Application	Version	FaFuzzer #crashes/bbls/inputs	VUzzer #crashes/bbls/inputs	AFLPIN #crashes/bbls/inputs	#Bugs
TIFF	libtiff	4.0.10	54/5896/10.8K	0/4338/21.2K	0/2876/32K	1
TIFF	exiv2	0.27.99	128/3736/8.5K	19/3215/13.3K	10/1978/31.5K	2
JPG			39/4230/13.6K	5/3108/25K	0/2553/41.8K	1
MP3	libav	12.3	62/1238/7.2K	11/992/8.8K	0/642/29.3K	1
AVI			93/987/9.1K	36/824/10.2K	8/719/18.2K	1
ZIP	zziplib	0.13.69	12/6881/2.3K	2/5320/4.5K	0/3223/37.9K	1
ELF	binutils	2.32	81/2975/16K	23/2487/26.9K	15/1578/44.6K	0

This indicates that FaFuzzer can achieve most effective mutation, followed by VUzzer, and AFLPIN is worst. In addition, our evaluation on other target programs shows similar results. There are two reasons for this result. First, field-level mutation can reduce the damage to the seed inputs, thus generating more effective samples. Second, field-type based mutation can help find the more interesting values to replace the offsets in compare instructions or functions, thereby increasing code coverage. VUzzer also performs better then AFLPIN, because its DTA engine makes it generate more effective inputs, while AFLPIN's mutations are largely random.

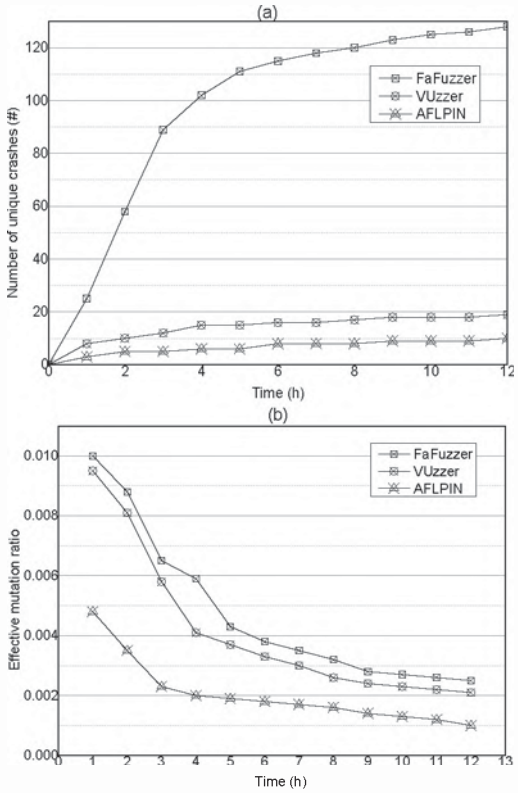

Figure 6. Performance comparison on exiv2. (a)Unique crashes found over time (b)Effective mutation ratio

V. CONCLUSION

In this paper, we elaborate on challenges faced by current evolutionary fuzzers while mutating seed inputs, that is, they are not aware of the input structure and mutate the input randomly, which greatly reduces the efficiency of fuzzing. To this end, we propose a field-aware mutation strategy that uses existed input specifications to extract input field and type information and distinguishes the importance of fields in the input by code assessment based on the vulnerability metrics. Then priority is given to important field and the mutation strategy is adjusted based on the field type when mutating. Finally, we implemented the prototype tool FaFuzzer and evaluated on datasets with known vulnerabilities and the applications with unknown vulnerabilities. The experimental results show that FaFuzzer can find more bugs with fewer but more effective inputs than the current advanced fuzzers such as VUzzer and AFLPIN, while achieving higher code coverage. In addition, we found many unknown bugs in real-world applications and reported them to the relevant vendors.

REFERENCES

[1] http://heartbleed.com/

[2] https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[3] https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[4] https://animal0day.blogspot.com/2017/07/from-fuzzing-apache-httpd-server-to-cve.html

[5] https://www.peach.tech/

[6] http://lcamtuf.coredump.cx/afl/

[7] https://github.com/samhocevar/zzuf

[8] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]//NDSS. 2016, 16(2016): 1-16.

[9] Zhao L, Duan Y, Yin H, et al. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing[C]//Proceedings of the Network and Distributed System Security Symposium. 2019.

[10] Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing[J]. arXiv preprint arXiv:1711.04596, 2017.

[11] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]//NDSS. 2017, 17: 1-14.

[12] https://github.com/mothran/aflpin

[13] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[14] https://www.exploit-db.com/exploits/39875

[15] https://www.sec-consult.com/en/blog/2017/09/hack-the-hacker-fuzzing-mimikatz-on-windows-with-winafl-heatmaps-0day/

[16] 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.

[17] 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.

[18] http://www.sweetscape.com/010editor/

[19] http://www.sweetscape.com/010editor/repository/templates/

[20] https://pfp.readthedocs.io/en/latest/

[21] https://github.com/idapython/src

[22] http://cve.mitre.org/

[23] http://bugzilla.maptools.org/show_bug.cgi?id=2848