Gen Zhang, Peng-Fei Wang, Tai Yue *et al.* *et al.* ovAFLow: Detecting Memory Corruption Bugs with Fuzzing-based Taint Inference. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY xxx: 1–22. This is a pre-print of an article published in Journal of Computer Science and Technology. The final authenticated version is available online at: https://doi.org/10.1007/s11390-021-1600-9.

ovAFLow: Detecting Memory Corruption Bugs with Fuzzing-based Taint Inference

Gen Zhang[1,], Peng-Fei Wang[1], Tai Yue[1], Xiang-Dong Kong[1], Xu Zhou[1], and Kai Lu[1], *Member, CCF*

[1] *College of Computer, National University of Defense Technology, Changsha 410073, China*

E-mail: {zhanggen, pfwang, yuetai17, kongxiangdong, zhouxu, kailu}@nudt.edu.cn
Received May 21, 2021; accepted November 15, 2021.

Abstract Grey-box fuzzing is an effective technology to detect software vulnerabilities, such as memory corruption. Previous fuzzers in detecting memory corruption bugs either use heavy-weight analysis, or use techniques which are not customized for memory corruption detection. In this paper, we propose a novel memory bug guided fuzzer, ovAFLow. To begin with, we broaden the memory corruption targets where we frequently identify bugs. Next, ovAFLow utilizes light-weight and effective methods to build connections between the fuzzing inputs and these corruption targets. Based on the connection results, ovAFLow uses customized techniques to direct the fuzzing process closer to memory corruption. We evaluate ovAFLow against state-of-the-art fuzzers, including AFL (american fuzzy lop), AFLFast, FairFuzz, QSYM, Angora, TIFF, and TortoiseFuzz. The evaluation results show better vulnerability detection ability of ovAFLow, and the performance overhead is acceptable. Moreover, we identify 12 new memory corruption bugs and two CVEs (common vulnerability exposures) with the help of ovAFLow.

Keywords fuzzing, memory corruption, taint inference

1 Introduction

Fuzz testing, or fuzzing, was introduced by Miller [1] in 1990. After three decades of development, fuzzing has been widely adopted both in research and industry to detect vulnerabilities and bugs.

Ever since the emergence of AFL (american fuzzy lop) [1], there has been an ongoing trend of coverage-guided grey-box fuzzing (CGF) techniques [2–6]. In essence, these CGF tools share similar core ideas that they are designed to cover as many program paths as possible. The key insight of driving the fuzzing process with code coverage is to cover the program paths of the program under test (PUT) and expose deeply hidden bugs.

Existing CGF tools treat all program paths equally and spend too much effort in increasing code coverage [7, 8]. They forget the crux of a fuzzer lies in the ability to effectively detect bugs. Besides code coverage, more guiding information is demanded to boost the fuzzing process, such as memory bug information. Previous work used this memory bug information, including TIFF [9], MemFuzz [10], and CollAFL [11]. TIFF uses dynamic taint analysis (DTA) to identify the in-

Regular Paper

This work was supported by the National High-level Personnel for Defense Technology Program under Grant No. 2017-JCJQ-ZQ-013, the Natural Science Foundation of China under Grant No. 61902405, the Natural Science Foundation of China under Grant No. 61902412, the PDL Research Foundation under Grant No. 6142110190404, and the Research Project of National University of Defense Technology under Grant No. ZK20-17.

©Institute of Computing Technology, Chinese Academy of Sciences 2021

[1] AFL. `http://lcamtuf.coredump.cx/afl`, Oct. 2021

put bytes that can affect the values of important target variables in the program. In MemFuzz and CollAFL, inputs with more memory accesses are executed with a higher possibility in the fuzzing campaign.

These fuzzers identify some memory corruption bugs. However, some inherent drawbacks reside in them. 1) First, they are short-sighted in recognizing the target variables in the programs, e.g., TIFF manually collects 17 memory operation functions, and their arguments are treated as targets. The detection method is not automatic, and it detects an insufficient number of functions and target variables. It is known to us that no matter what extreme values we can use in mutation, insufficient taint targets will always result in fewer memory corruption bugs. 2) DTA is too heavy-weight when adopted in fuzzing. Though complicated analysis, such as DTA, can provide precise information, it suffers from slow execution speed (the number of executions of the PUT every second). It is commonly acknowledged that fuzzing should be fast and light-weight, and complicated analysis techniques that can slow down the fuzzing process should be excluded [12–14]. Besides, DTA requires extensive manual effort to write platform-specific rules for every instruction. It is difficult to scale DTA to different platforms. 3) MemFuzz and CollAFL use coarse-grained seed prioritization by counting the number of memory accesses in a seed. The guiding information is not fine-grained enough for memory corruption detection. Memory bugs are closely related to sensitive memory operations. Therefore, simply counting the number of memory accesses is not an ideal solution.

In this paper, we propose ovAFLow to overcome the above drawbacks of existing memory bug guided fuzzers. Fundamentally, our primary intention is to identify more memory corruption with less performance overhead. 1) We broaden the taint targets from two perspectives. First, we automatically identify memory operation functions from real-world programs through static analysis and treat their arguments as targets. Second, loops are commonly recognized as vulnerable program parts [8, 15, 16]. Therefore, we identify loops with memory accesses and treat the variables that control the number of iterations of the loop as taint targets. 2) To achieve acceptable fuzzing speed, we adopt fuzzing-based taint inference (FTI) to obtain taint information in ovAFLow. FTI is a newly proposed technique, which can get taint information during the fuzzing process without suffering from performance overhead [17–19]. The basic idea of this technique is to monitor the taint targets after mutating the input bytes. If the value of the target changes after the mutation, we say that the input can taint the target. Furthermore, FTI is free of any intensive manual effort to write the platform-specific taint rules in DTA. 3) We design a fine-grained seed prioritization strategy that contributes to the bug detection of the fuzzing process. Taint information is used in our strategy. We prioritize inputs that contain more identified taint input bytes. The intuition is that the more bytes in the input that can taint the target variables, the more chances to trigger memory corruption. For example, we prioritize input A with three taint bytes over input B with one. Compared with simply counting the number of memory accesses, our solution can guide the fuzzing process to more sensitive memory operations and closer to memory corruption.

We implement the prototype of ovAFLow based on AFL. To answer the research questions in Section 5, we evaluate ovAFLow against state-of-the-art fuzzers in real-world programs and the LAVA-M [20] dataset. Our evaluations show that ovAFLow identifies more program crashes in 67 out of the 72 comparison experiments with performance overhead at around 10%.

Therefore, ovAFLow achieves the goal of detecting more memory corruption bugs with acceptable performance overhead. Moreover, we expose 12 new memory corruption bugs and two CVEs (common vulnerability exposures) in real-world programs. This result also confirms the bug detection ability of ovAFLow.

In conclusion, we make the following contributions:

- 1) We reveal the defects of taint targets of previous memory bug fuzzers and expand them to memory operation function arguments and memory access loop counts.
- 2) Realizing the inappropriateness of heavy-weight taint analysis in previous fuzzing tools, we use FTI to identify taint input bytes that can taint our targets with acceptable performance overhead.
- 3) We propose customized mutation and seed prioritization strategies based on the taint information, guiding the fuzzing process to more sensitive memory operations and closer to memory corruption than previous work with coarse-grained prioritization strategies.
- 4) Extensive evaluations are performed to compare ovAFLow with state-of-the-art fuzzers in real-world programs and the LAVA-M dataset, and the results show that ovAFLow completes the task of detecting more memory bugs with acceptable overhead.

The rest of the paper is organized as follows. Section 2 discusses about the background. Section 3 contains the technique details. Implementation details are listed in Section 4. Section 5 shows the evaluation results. The related work is in Section 6. Section 7 concludes this paper.

2 Background

2.1 Fuzzing and AFL

Fuzzing is a widely used software testing technique. It can automatically generate inputs to expose vulnerabilities. Since the introduction of AFL in 2013, researchers have been focusing on coverage-guided greybox fuzzers. CGF requires simple program analysis, such as compile-time instrumentation, to obtain feedback and guide the fuzzing process. However, its counterparts, such as black-box and white-box fuzzers, either have no execution feedback to evolve the process, or require heavy-weight analysis techniques. CGF tools outperform black-box and white-box fuzzers in both efficiency and effectiveness because of the perfect balance between the fuzzing speed and accuracy. Therefore, we implement our ovAFLow prototype based on the AFL framework.

Fig. 1 is the basic workflow of AFL. When we start AFL, 1) it reads all the initial seeds to the seed queue. 2) Based on the prioritization rules, AFL selects a seed as an input. 3) The input is mutated to generate several new inputs. 4) AFL executes the PUT using these inputs and 5) monitors whether a new program path is covered. 6) If an input can cover a new path, AFL adds it to the seed queue. Otherwise, the input is discarded. 7) AFL goes back to step 2) and continues to fuzz the PUT. In the following parts, we are going to discuss the mutation and seed prioritization strategies in AFL.

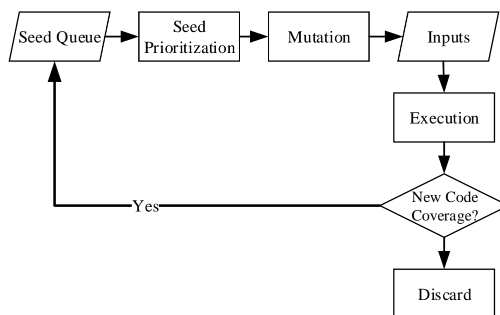


Fig.1. Basic workflow of AFL.

2.1.1 Mutation

The mutation strategy of AFL can be divided into two parts: the deterministic stage and the havoc stage. The deterministic stage includes “bitflip”, “arithmetic”, “interest”, and “dictionary”. “havoc” and “splice” make up the havoc stage. Table 1 shows the basic procedure of each mutation strategy. For example, for an input “010”, “bitflip-1” mutates it to “110”, “000”, and “011”. Specifically, we integrate the FTI engine into the “bitflip-1” stage. Therefore, ovAFLow will not perform extra executions to obtain the taint information.

Table 1. Mutation Strategies of AFL

Item	Detail
bitflip	Flip by bit, one becomes zero, and zero becomes one
arithmetic	Integer addition or subtraction and arithmetic operation
interest	Replace some special integers in the original input
dictionary	Replace or insert the tokens automatically generated or provided by the user into the original input
havoc	Contain multiple rounds of variation of the original input, and each round is a combination of a variety of ways
splice	The two seed files are spliced to get a new file, and havoc mutation is performed on the new file

2.1.2 Seed Prioritization

This part can also be divided into two components. First, AFL adopts input filtering to collect “interesting” seeds which can cover new program paths, or the hit count of a path reaches a new scale. For instance, input A covers *Path1* three times, input B covers *Path1* and *Path2*, and input C covers *Path1* 100 times. In this case, both input B and C are interesting in the input filtering of AFL. Next, AFL uses queue culling to rank the seeds. This algorithm prefers to prioritize inputs with a smaller size and faster execution speed. After queue culling, a subset of the inputs is selected which is more efficient and maintains the original code coverage.

2.2 Fuzzing-based Taint Inference

Generally speaking, complicated taint analysis techniques, such as DTA, are hardly suitable when adopted in a fuzzing situation. The major differences between DTA and FTI are illustrated in Table 2. Most significantly, DTA suffers from performance overhead. Whereas for FTI, it can be fast, and our evaluation shows its overhead is about 10%. Meanwhile, DTA requires extensive manual effort to write the taint rules, and these rules are specific for different instructions of different platforms. FTI requires no specific rule. As for accuracy, FTI has no over-taint issue. If FTI reports an input byte can taint the targets, it is very likely to be true. However, under-taint is ubiquitous in DTA because of the implicit data flow or loss of information. FTI is free of these issues. For the above reasons, we adopt FTI to assist our fuzzer to get taint information with low performance overhead.

Table 2. Differences Between DTA and FTI

Item	DTA	FTI
Overhead	High	Low
Manual effort	High	Low
Platform-specific	Yes	No
Over-taint	Yes	No
Under-taint	High	Low

3 Technique Details

3.1 Overview

Fig. 2 shows the overview of ovAFLow, including taint target recognition, FTI, mutation, seed prioritization, and the main fuzzing loop. The colored shapes denote the changes to the original AFL. The taint target recognition identifies memory operation function arguments and memory access loop counts. Then, the main fuzzing loop starts to execute the PUT. Receiving the taint targets, the FTI engine detects the input bytes that can taint the targets during program execution.

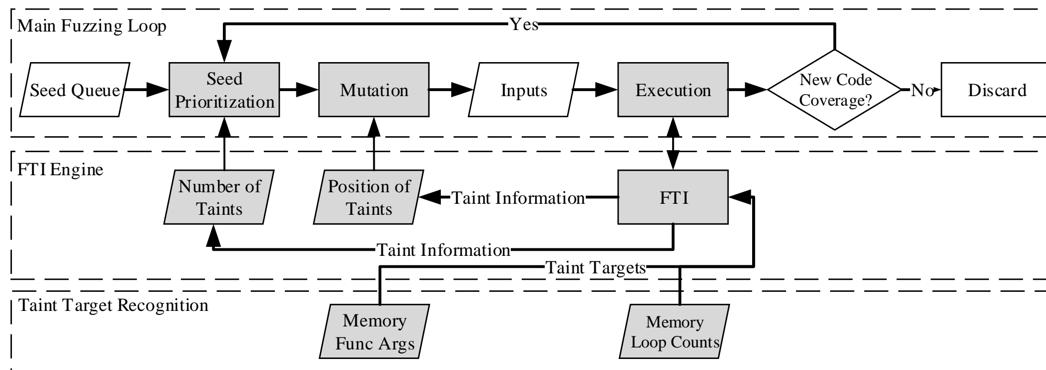


Fig.2. Overview of ovAFLow.

Meanwhile, the positions and the number of the taint bytes are delivered to mutation and seed prioritization, respectively, helping the fuzzing process to expose more memory corruption bugs.

3.2 Taint Target Recognition

As mentioned above, TIFF manually collects 17 library functions and treats their arguments as taint targets. We argue that this solution is too narrow-scoped, and it misses commonly used memory operation functions, such as `CRYPTO_malloc`(). These absent functions are also vulnerable. Incomplete recognition of taint targets will lead to the detection of fewer memory corruption bugs.

To solve this issue, we come up with two aspects of methods to obtain the taint targets. The first one is extending the number of memory operation functions by automatically detecting them from real-world applications. With the help of static analysis, our basic idea is to set up rules to identify functions that satisfy our heuristics. Then, we can use statistical methods to confirm these memory operation functions, e.g., we can use the number of times a function is identified as a memory operation function. To achieve this goal, two questions need to be answered. 1) What real-world programs can we use to extract these functions? 2) What heuristics can we use to filter these functions?

1) To extract our desired functions, the real-world programs need to be sufficient in the amount of code to perform statistical analysis, and the programs should contain utilities of various kinds of purposes to maintain diversity. Through our study of commonly used datasets, we finally choose the Google fuzzer test suite [2] as our target. This dataset contains over six million lines of code and 21 programs of different kinds, such as json and libpng. 2) We set up two heuristics to filter the functions. One is whether this function is performing memory operations, and the other is whether the arguments contain an integer that is similar to the `size` argument in `memcpy`(). These two heuristics ensure that we obtain memory operation functions, and they are controllable through the `size` argument. By mutating the input bytes controlling the `size`, we can trigger memory corruption with a higher possibility.

[2] Google fuzzer test suite. `https://github.com/google/fuzzer-test-suite`, Oct. 2021.

Algorithm 1 Memory Operation Function Identification.

```
Require: Dataset {DS}
 1: {MF} = ∅
 2: for F in {DS} do
 3:     f_F = 0
 4: end for
 5: for F in {DS} do
 6:     if mem_access(F) == True then
 7:         if type_args(F) == int then
 8:             f_F = f_F + 1
 9:         end if
10:     end if
11: end for
12: for F in {DS} do
13:     if f_F >= Threshold then
14:         {MF} = F∪{MF}
15:     end if
16: end for
Ensure: Memory operation function arguments {MF}
```

As shown in Algorithm 1, the input of the memory operation function identification is the dataset *DS*. The outputs are the arguments of the identified functions. First, as shown in lines 2-4, for every function in the dataset, we declare a frequency variable f and set f to zero at the beginning. Then, in lines 5-11, we check every call site of function F. We examine whether F is accessing memory and whether the arguments of F contain a `size` argument. If both the conditions are satisfied, we increase f_F by one. It means this call site of F is identified as a memory operation function. After examining all the source code in the dataset, we determine whether F is a memory operation function by a threshold, which is shown in lines 12-15. If f_F is greater or equal to the threshold, we add F to the set of memory operation functions.

In addition, the threshold is determined by the following steps. When we have the frequencies of all the functions, we can get the statistics of the frequencies. The threshold is based on the statistics. For example, we can calculate the average number of all the frequencies and set the threshold to the average number.

The second part of taint target recognition is memory access loop count identification. Loops with memory accesses are vulnerable sections of programs, which can lead to memory corruption. Our intuition is to go beyond the scope of function arguments of traditional approaches and to broaden the taint targets to the variables which control the loop iterations, i.e., loop counts. When the input bytes that taint the loop counts are mutated to extreme values, memory corruption bugs are triggered. First, we construct control flow graphs (CFG) of the PUT and then identify loops with standard back edge [21] analysis. Next, we filter out loops without memory accesses or loop counts and finally get our desired memory access loops and counts.

Algorithm 2 Memory Access Loop Count Identification.

```
Require: Target programs PUT
 1: {ML} = ∅
 2: CFG = build_cfg(PUT)
 3: {Loop} = back_edge(CFG)
 4: for L in {Loop} do
 5:     if mem_access(L) == True then
 6:         if have_count(L) == True then
 7:             {ML} = L∪{ML}
 8:         end if
 9:     end if
10: end for
Ensure: Memory access loop counts {ML}
```

As shown in Algorithm 2, the inputs of the process are the target programs, i.e., the PUTs. The outputs are the identified memory access loop counts. Lines 2-3 show the process of constructing the CFG and identifying loops. In lines 4-10, we check each loop to determine whether it has memory accesses and whether the loop iterations are controlled by a variable, i.e., the loop count. If the conditions are satisfied, we add this loop count to the set of loop counts.

3.3 Fuzzing-based Taint Inference

Once we obtain the taint targets in the PUT, we can start the FTI engine to build connections between the input bytes and these taint targets.

Algorithm 3 Fuzzing-based Taint Inference.

Require: Function arguments $\{MF\}$ and loop counts $\{ML\}$
1: $\{T\} = \emptyset$
2: cksum1 = `cksum`($\{MF\}, \{ML\}$)
3: **for** $byte_i$ in Input_bytes **do**
4: `mut_exe`($byte_i$)
5: $cksum2$ = `cksum`($\{MF\}, \{ML\}$)
6: **if** $cksum1! = cksum2$ **then**
7: $\{T\} = byte_i \cup \{T\}$
8: **end if**
9: **end for**
Ensure: Taint input bytes $\{T\}$

As shown in Algorithm 3, the inputs contain the memory operation function arguments and memory access loop counts. The outputs are the taint input bytes. Before mutating $byte_i$, we calculate the checksum of the function arguments and the loop counts. Then, $byte_i$ is mutated, and we execute the PUT to calculate the new checksum. If the two checksums are not equal, it means the values of the taint targets are changed, and mutating $byte_i$ can cause this change. Therefore, we add $byte_i$ to the set of taint input bytes.

Fig. 3 shows a working example of FTI. Assume that we have four bytes in the input and three taint targets. Every time we mutate one byte from "00" to "01", we monitor the changes in the targets. For instance, when we mutate *byte1* from "00" to "01", the value of *Var1* changes from zero to 16, and we can say that *byte1* can taint *Var1*. When all the mutations finish, we will unite the results in each step to handle issues where several continuous bytes taint the same target.

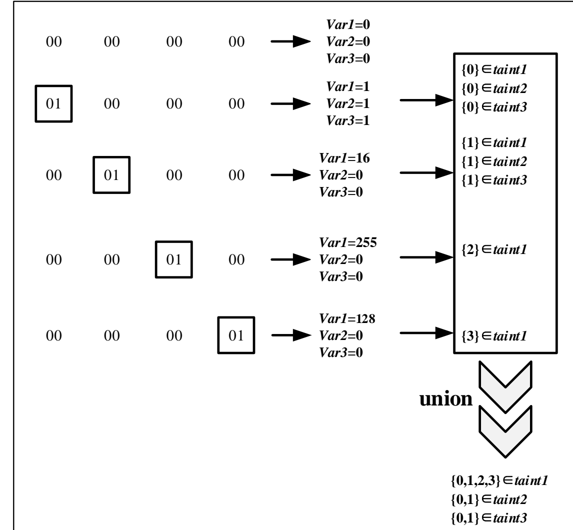


Fig.3. Basic procedure of FTI.

3.4 Mutation and Seed Prioritization Strategies.

3.4.1 Mutation

In this part, we are going to solve two problems. 1) Where to perform the memory bug guided mutation? 2) What values to replace with? As mentioned above, FTI identifies input bytes that can taint the targets, and these input bytes are delivered to mutation. Receiving the positions of taint input bytes, our mutation engine can mutate these bytes to extreme values. For memory operation function arguments, these values will cause manipulation of an unexpected amount of memory, resulting in memory corruption. As for memory access loop counts, by increasing the iterations of accessing memory, we can also trigger sensitive memory operations. Nevertheless, the extreme values are not randomly selected. We manually analyze numerous real-world memory corruption bugs and collect commonly seen values in Table 3.

Table 3. Manually Collected Extreme Values for Triggering Memory Corruption

Extreme Value	Description
-2147483648LL	Overflow signed 32-bit when decremented
-100663046	Large negative number (endian-agnostic)
-32769	Overflow signed 16-bit
32768	Overflow unsigned 16-bit
65535	Overflow unsigned 16-bit when incremented
65536	Overflow unsigned 16 bit
100663045	Large positive number (endian-agnostic)
2147483647	Overflow signed 32-bit when incremented
2147483631	0x7fffffef
2147483646	0x7ffffffe
2147483648	0x80000000
2147483663	0x8000000f
4294967294	0xfffffffe
4294967295	0xffffffff

This table contains 14 extreme values for mutation. By replacing the taint input bytes with these values, memory corruption bugs are triggered with a higher probability. For instance, when we replace four bytes in the input with the 32-bit overflow value, the memory operation function and memory access loop may operate an overflowed amount of memory, and this can cause memory corruption.

3.4.2 Seed Prioritization

Previous work in memory bug guided fuzzing simply used the number of memory accesses to prioritize seeds. We argue that this heuristics is not effective enough, because it fail to focus on more sensitive memory operations that may easily trigger memory corruption. By realizing this problem, we propose a more fine-grained seed prioritization strategy, aiming to prioritize seeds with more taint input bytes. The input bytes can taint the memory operation function arguments and memory access loop counts. Concentrating on these scenarios rather than simply the number of memory accesses will help the fuzzer to trigger more memory corruption bugs. In addition, we still keep the original coverage-based seed prioritization strategy of AFL to cover as many program paths as possible. In conclusion, our seed prioritization strategy is shown in the following equation.

$$Prioritize[seed_i] = \begin{cases} 1 & \text{if } taint_i > taint_j \\ & or\ afl_prio_i > afl_prio_j, \\ 0 & \text{otherwise.} \end{cases}$$

$taint_i$ and afl_prio_i denote the strategies of ovAFLow and AFL, respectively. $seed_i$ is the current seed, and $Prioritize[]$ determines whether this seed should be prioritized. As shown in the equation, a seed that has more taint bytes or covers more program paths can be prioritized. Otherwise, it will not be prioritized.

Algorithm 4 Seed Prioritization and Mutation.

Require: Taint input bytes $\{T\}$ and seeds $\{S\}$
1: $s =$ `prioritize`$(\{T\}, \{S\})$
2: **for** t in $\{T\}$ **do**
3: $s' =$ `mutation`(s, t)
4: **end for**
Ensure: Mutated seed s'

Algorithm 4 shows the procedure of our seed prioritization and mutation strategies. The inputs are the identified taint bytes and the seeds. The output is the mutated seed. In line 1, we select the most favored seed among all the seeds through seed prioritization. Lines 2-4 show the process of mutation, where we mutate the taint input bytes in the seed to extreme values. This mutated seed will be executed in the PUT in the next round of fuzzing.

4 Implementation Details

In this section, we discuss about the details of the implementation of ovAFLow, including the memory operation function identification, the FTI engine, and other components. The ovAFLow prototype is released[3].

4.1 Memory Operation Function Identification

We write LLVM passes to conduct static analysis to finish this task. By examining the functions

[3]ovAFLow prototype. `https://github.com/zhanggenex/ovAFLow.git`, Oct. 2021.

and their arguments, we can identify whether this function is operating on memory and whether a `size` argument exists. We use `mayReadFromMemory`() and `mayWriteToMemory`() of LLVM to determine whether this function is accessing memory. In addition, we use `getType`() in LLVM to get the type of the arguments.

4.2 Taint Target Instrumentation

After we obtain the function arguments and loop counts in the PUT, we will trace their values and changes. We modify the $\texttt{afl} - \texttt{llvm} - \texttt{pass.so.cc}$ in AFL, which instruments the PUT and records the values in a new bitmap every time these targets are met. In detail, we use the shared memory data structure, i.e., bitmap, to store the values of the taint targets. In instrumentation, we first declare a pointer variable `TaintPtr` for the bitmap region. Next, we locate the taint targets in the source code, i.e., the function arguments and loop counts. The values of the taint targets are instrumented, which are stored in the bitmap. In addition, the changes in the checksums of the bitmap can represent the changes in the bitmap. In this way, every time the PUT is executed, we can monitor the values of the taint targets through the checksums of the bitmap.

4.3 FTI

We integrate the FTI engine into the "bitflip-1" stage of AFL and monitor the changes in our taint targets. For example, after input A is executed, the checksum of the bitmap is *cksum1*, and input B results in *cksum2*. We compare *cksum1* with *cksum2* to identify whether the values of our taint targets are changed. If the checksums are not equal, the mutated byte can taint the taint targets.

4.4 Mutation to Extreme Values

To insert the extreme values in Table 3 into the inputs, we modify the "interest-32" stage of AFL. The FTI engine tells us the positions of the taint bytes in the input, and we replace these bytes with extreme values to trigger memory corruption.

4.5 Seed Prioritization

We modify the corresponding code that controls the prioritization of seeds in AFL. AFL maintains a `top_rate` data structure to get the most favored seed. Besides file size and execution speed, we also use the number of taint bytes in the input as one factor to calculate the `top_rate` score to prioritize seeds.

5 Evaluation

In this section, we are going to answer these research questions.

- RQ1: Can ovAFLow find more unique crashes than state-of-the-art fuzzers?
- RQ2: Can ovAFLow identify more memory corruption bugs?
- RQ3: Can ovAFLow accomplish the vulnerability detection task with acceptable performance overhead?
- RQ4: Are the mutation and seed prioritization strategies in ovAFLow effective?

5.1 Setup

All our evaluations are conducted on a server with 48 cores of Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 128GB of RAM, and a Linux kernel of 4.4.0-142-generic. The evaluations are divided into two parts: the real-world programs and the LAVA-M dataset [20]. The information of the target programs and the baseline fuzzers will be given in the following subsections.

5.2 Real-world Programs

We test 12 real-world programs in total. They include image processing programs (tiff2pdf and tiff2ps from libtiff, and exiv2 from exiv2), multimedia programs (mp42aac and mp4tag from Bento4, and avconv from libav), pdf programs (podofopdfinfo and podofotxtextract from podofo), xml programs (xmllint from libxml), text processing program (infotacap from ncurses), and binary processing programs (nm and readelf from Binutils). Moreover, Table 4 shows the basic information of these target programs.

Table 4. Target Programs

Targets	Version	Input Format
mp42aac @@ a.aac	Bento4-1.5.1-628	mp4
mp4tag –show-tags –list-symbols –list-keys @@	Bento4-1.5.1-628	mp4
tiff2pdf @@	libtiff-4.0.7	tiff
tiff2ps @@	libtiff-4.0.7	tiff
podofopdfinfo @@ (pdfinfo)	podofo-0.9.6	pdf
xmllint @@	libxml-2.98	xml
exiv2 @@ /dev/null	exiv2-0.27	jpeg
infototap @@	ncurses-6.1	txt
avconv -y -i @@ -f null	libav-12.3	mp4
podofotextextraxt @@ (pdfext)	podofo-0.9.6	pdf
nm -C @@	Binutils-2.30	elf
readelf -a @@	Binutils-2.30	elf

For real-world programs, AFL, AFLFast [2], FairFuzz [6], TortoiseFuzz [8], QSYM [5], and Angora [4] are used in our evaluation [4]. They are chosen because they are state-of-the-art fuzzers, and recent fuzzing papers frequently used these fuzzers as baselines [8, 17, 22–24].

We use seeds in the `testcase` directory provided by AFL as initial seeds. We use the number of unique crashes discovered by each fuzzer as the first metric to answer RQ1. The second metric is the number of memory corruption bugs in the crashes to answer RQ2. Moreover, to answer RQ3, we compare the execution speed of ovAFLow with the baseline fuzzers. RQ4 is answered with the number of crashes triggered by our mutation and seed prioritization strategies among all the crashes. All the evaluations in this subsection are repeated 10 times for 24 hours to eliminate the randomness during fuzzing, and the p values of the Mann-Whitney (M-W) U test are given to show the significance of the differences of the evaluations. Besides, the bold numbers in the evaluation results are the best among all the results.

Table 5. Unique Crashes Discovered by the Fuzzers

Targets	OA	AF	AT	FF	TF	QS	AG
mp42aac	217.9	134.0	185.6	279.8	188.8	**299.8**	114.1
mp4tag	229.3	202.2	186.6	**306.0**	199.2	199.9	202.8
tiff2pdf	**25.1**	7.8	1.4	4.5	5.5	1.5	8.7
tiff2ps	**22.6**	19.2	15.8	18.5	13.1	16.4	18.2
pdfinfo	**30.3**	10.2	22.1	25.3	6.3	9.8	12.7
xmllint	**3.3**	0.5	0.0	0.0	0.0	0.0	0.0
exiv2	**57.6**	39.1	44.5	43.9	44.9	33.4	41.2
infotocap	273.9	159.9	264.0	**300.7**	113.1	209.1	197.6
avconv	226.4	34.3	38.5	377.1	134.9	44.9	**378.9**
pdfext	**80.3**	69.5	62.9	73.7	71.4	61.1	64.1
nm	**10.1**	0.8	3.0	0.0	0.0	0.0	4.5
readelf	**87.3**	67.0	67.0	77.3	66.1	61.9	71.9

Table 6. p Values of Table 5

Targets	AF	AT	FF	TF	QS	AG
mp42aac	9.03e-5	1.62e-4	0.99	2.35e-4	0.99	9.03e-5
mp4tag	2.28e-3	1.29e-4	0.99	1.38e-4	1.45e-4	3.55e-3
tiff2pdf	8.93e-5	7.25e-5	8.58e-5	8.61e-5	7.58e-5	1.59e-4
tiff2ps	2.37e-4	5.19e-5	5.21e-4	4.03e-5	7.29e-5	1.44e-4
pdfinfo	1.12e-4	0.11	0.04	9.03e-5	1.01e-4	5.66e-4
xmllint	0.03	2.92e-3	2.92e-3	2.92e-3	2.92e-3	2.92e-3
exiv2	1.01e-4	9.35e-4	0.02	0.03	9.03e-5	1.02e-3
infotocap	4.33e-4	0.39	0.51	9.03e-5	0.01	2.05e-3
avconv	9.03e-5	9.03e-5	0.99	0.02	5.87e-3	0.99
pdfext	0.01	7.70e-3	0.24	0.02	5.21e-3	9.33e-3
nm	3.28e-3	0.03	3.72e-4	.72e-4	3.72e-4	0.04
readelf	2.88e-4	1.81e-4	0.01	9.03e-5	1.22e-5	3.78e-3

5.2.1 *Unique Crashes*

Table 5 and Table 6 show the number of average unique crashes of 10 repeated runs and the p values of the M-W U test, respectively. The number of unique crashes a fuzzer can find is an important indicator of the vulnerability detection ability. In total, among the 72 pairs of comparisons, ovAFLow triggers more unique crashes than the competitors 67 times. Especially in xmllint and nm, ovAFLow exposes unique crashes and

[4]We use these abbreviations in this paper: OA=ovAFLow, AF=AFL, AT=AFLFast, FF=FairFuzz, TF=TortoiseFuzz, QS=QSYM, AG=Angora.

other fuzzers cannot. Whereas for the p values of the M-W U test, we can see 64 pairs of comparisons with a significant difference ($p < 0.05$) in the 67 cases. This indicates that ovAFLow can trigger more unique crashes in more than 95% comparison evaluations with a significant difference.

ovAFLow outperforms AFL, AFLFast, and TortoiseFuzz in all the evaluations, demonstrating better vulnerability detection ability of ovAFLow. However, there are four pairs of comparisons where other fuzzers identify more crashes. We argue that the reason behind this is the rare branches FairFuzz, QSYM, and Angora can cover, and ovAFLow is not focusing on these scenarios. Nevertheless, ovAFLow still outperforms FairFuzz in eight out of the 12 comparisons, outperforms QSYM in 11 out of the 12 comparisons, and outperforms Angora in 11 out of the 12 comparisons. In conclusion, we can answer RQ1 that ovAFLow identifies more unique crashes than state-of-the-art fuzzers.

Table 7. Memory Corruption Bugs among all the Unique Crashes

Targets	OA	AF	AT	FF	TF	QS	AG
mp42aac	3.8	2.6	**21.7**	4.4	3.1	2.2	2.2
mp4tag	**4.6**	2.8	3.5	3.9	2.5	1.9	1.9
tiff2pdf	**1.8**	0.0	0.0	0.0	0.0	0.0	0.0
tiff2ps	**21.9**	18.8	15.5	16.5	20.1	18.5	15.1
pdfinfo	**19.8**	3.5	9.8	10.6	11.8	4.3	10.1
xmllint	0.0	0.0	0.0	0.0	0.0	0.0	0.0
exiv2	9.5	5.1	7.1	19.4	**19.9**	4.8	6.0
infotocap	171.25	88.9	138.7	230.9	**244.5**	89.1	98.2
avconv	3.8	0.0	0.1	**11.8**	3.5	0.1	0.0
pdfext	**79.6**	67.5	60.6	72.8	74.3	61.9	62.5
nm	**9.2**	0.8	3.0	0.0	3.1	0.0	0.0
readelf	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 8. p Values of Table 7

Targets	AF	AT	FF	TF	QS	AG
mp42aac	0.01	0.99	0.91	0.04	8.21e-3	7.55e-3
mp4tag	0.29	0.29	0.46	0.14	0.09	0.09
tiff2pdf	0.03	0.03	0.03	0.03	0.03	0.03
tiff2ps	1.82e-3	6.95e-05	8.96e-05	0.01	1.55e-3	5.69e-5
pdfinfo	7.64e-05	1.83e-3	5.39e-3	0.03	9.03e-5	2.88e-3
xmllint	-	-	-	-	-	-
exiv2	3.32e-3	0.14	0.71	0.82	2.12e-3	9.11e-3
infotocap	1.03e-3	0.02	0.89	0.91	5.41e-3	0.01
avconv	1.14e-4	2.52e-4	0.94	0.04	2.52e-4	1.14e-4
pdfext	5.54e-3	6.97e-3	0.22	0.35	9.21e-3	0.01
nm	7.60e-3	0.11	1.18e-4	0.25	1.18e-4	1.18e-4
readelf	-	-	-	-	-	-

5.2.2 Memory Corruption Bugs

Our design intention is to enable ovAFLow to find more memory corruption bugs. Therefore, we are going to demonstrate it with Table 7 and Table 8. Table 7 shows the number of memory corruption bugs among all the unique crashes in Table 5. We determine whether the crash is a memory corruption bug through manual analysis with the help of AdressSanitizer (ASAN) [25]. In general, ovAFLow triggers more memory bugs in 53 out of the 60 (there are 12 pairs where all fuzzers find no memory bug) pairs of comparisons. In addition, Table 8 shows the p values of the M-W U test. The results show ovAFLow outperforms others in 42 out of the 53 comparisons with a significant difference. Additionally, ovAFLow detects memory corruption in PUTs where others fail to, such as tiff2pdf.

Moreover, ovAFLow has better performance in 100%, 90%, 70%, 80%, 100%, and 100% of the pairs than AFL, AFLFast, FairFuzz, TortoiseFuzz, QSYM, and Angora, respectively. None of them use memory bug guided mutation or seed prioritization strategies which are used in ovAFLow. Therefore, they cannot trigger as many memory corruption bugs as ovAFLow. With this evaluation result, we can answer RQ2 that ovAFLow detects more memory bugs than the baseline fuzzers.

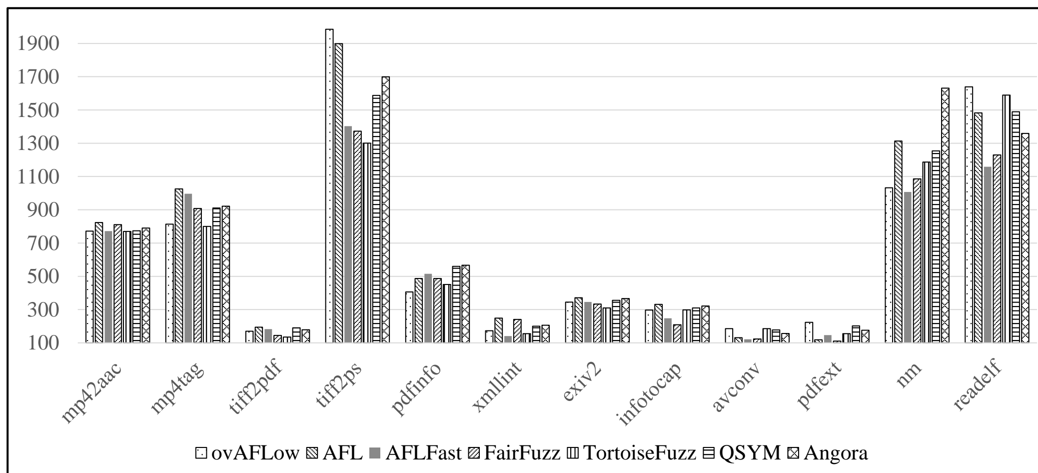


Fig.4. Average execution speed of each fuzzer during the 10 runs. The X-axis denotes the PUT, and the Y-axis is the number of executions per second.

Table 9. Average Execution Speed of Each Fuzzer

Fuzzer	Execution Speed
ovAFLow	670.3
AFL	702.3(-4.6%)
AFLFast	586.6(+14.3%)
FairFuzz	588.0(+14.0%)
TortoiseFuzz	611.5(+9.6%)
QSYM	667.6(+0.4%)
Angora	697.7(-3.9%)

Note: The numbers in the brackets show the performance increase or decrease compared with the baseline fuzzers.

5.2.3 Execution Speed

Fig. 4 shows the execution speed of each fuzzer. At first sight, ovAFLow has the same level of execution speed as the baseline fuzzers. In tiff2ps, avconv, podofotxtextract, and readelf, ovAFLow even achieves the highest speed. Specifically, we can see in 37 out of the 72 comparisons ovAFLow runs slower than others. Among the 37 pairs of comparisons, the average performance overhead is 10.3%.

Furthermore, we record the average speed of all the evaluations in Table 9. According to it, ovAFLow is only slower than AFL and Angora with less than 5% overhead. ovAFLow is faster than AFLFast, FairFuzz, and TortoiseFuzz with more than 9% performance increase. Moreover, GREYONE [17] is a state-of-the-art fuzzer using FTI. The authors claimed less than 25% performance overhead in the paper. Therefore, RQ3 is answered through this part of the evaluation. Compared with these state-of-the-art fuzzers, ovAFLow accomplishes the vulnerability detection task with acceptable performance overhead.

Table 10. Average Path Coverage of Each Fuzzer

Fuzzer	Path Coverage
ovAFLow	4113.9
AFL	3595.23(+14.4%)
AFLFast	3107.7(+32.4%)
FairFuzz	3665.9(+12.2%)
TortoiseFuzz	3556.4(+15.7%)
QSYM	4284.0(-4.0%)
Angora	4481.7(-8.2%)

Note: The numbers in the brackets show the increase or decrease compared with the baseline fuzzers.

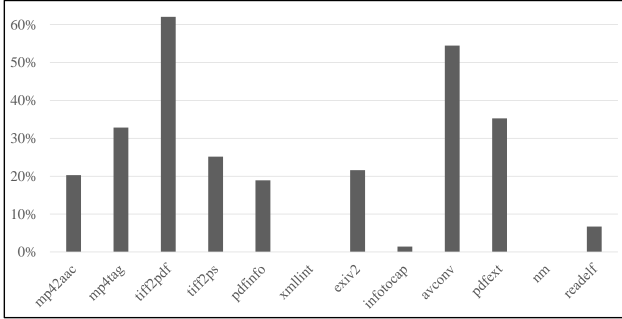


Fig.5. Percentages of crashes triggered by our mutation and seed prioritization strategies.

Table 11. Numbers of Crashes Triggered by Our Mutation and Seed Prioritization

PUT	Number of Crashes
mp42aac	44.0
mp4tag	74.7
tiff2pdf	13.7
tiff2ps	5.7
pdfinfo	5.5
xmllint	0.0
exiv2	12.4
infotocap	3.5
avconv	121.0
pdfext	28.3
nm	0.0
readelf	5.9

5.2.4 *Crashes Triggered by the Mutation and Seed Prioritization Strategies*

We propose memory bug guided mutation and seed prioritization in ovAFLow. Fig. 5 and Table 11 show the percentages and numbers of crashes triggered by these strategies in each PUT, respectively. As we can see in tiff2pdf, more than 60% of the crashes are triggered by our customized strategies. Furthermore, the average percentage of triggered crashes is around 25%, which means a quarter of all the crashes result from the mutation and seed prioritization strategies. The rest of the crashes are from the original strategies in AFL, which aim to improve code coverage. Therefore, we can answer RQ4 that our mutation and seed prioritization strategies make up 25% of all the unique crashes.

5.2.5 *Path Coverage*

Fig. 6 illustrates the code coverage of each fuzzer. We can see that ovAFLow covers more paths than the baseline fuzzers in at least five PUTs, such as nm. Though we are not targeting code coverage in ovAFLow, our mutation and seed prioritization strategies possibly help the fuzzing process cover more paths. Table 10 shows the average path coverage of all the evaluations. Compared with the baseline fuzzers, ovAFLow covers 14.4%, 32.4%, 12.2%, and 15.7% more average path than AFL, AFLFast, FairFuzz, and TortoiseFuzz, respectively. ovAFLow mutates the taint input bytes to extreme values, and these values possibly help cover more program paths. However, QSYM and Angora outperform ovAFLow in detecting program paths. Both QSYM and Angora are designed to pass magic bytes and solve the constraints in the programs. Therefore, they can outperform ovAFLow.

5.3 LAVA-M Dataset

Table 12. Number of Identified LAVA-M Bugs for Each Fuzzer

Target	Total	OA	QS	AG	TF	AT	FF	TF
base64	44	44+3	44	44+1	38	44	44	44
md5sum	57	57+3	57	57	27	57	57	57
uniq	28	28+1	28	28	28	28	28	28
who	2136	1724+14	1215	1866	171	1591	1699	1650

Note: The values consist of listed bugs and unlisted memory corruption bugs.

The LAVA-M dataset contains base64, md5sum, uniq, and who. They are manually injected with bugs that fuzzers need to pass numerous magic byte checks to trigger. In total, there are 44 injected bugs in base64, 57 in md5sum, 28 in uniq, and 2136 in who.

We select QSYM [5], Angora [4], TIFF [9], AFLFast, FairFuzz, and TortoiseFuzz to compare with. We run ovAFLow in one thread and QSYM in another. Adopting QSYM to help a fuzzer solve magic bytes in the LAVA-M dataset is commonly seen in previous work

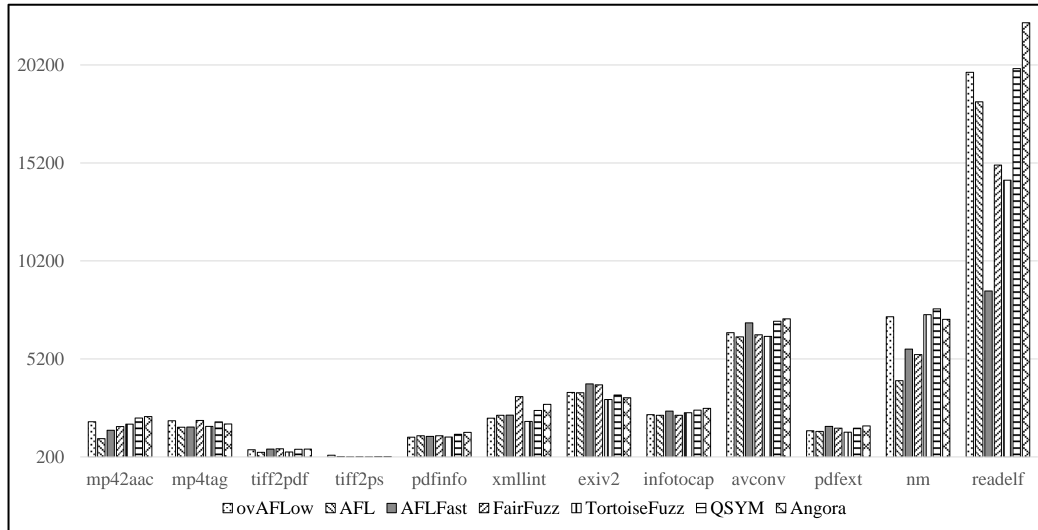


Fig.6. Average path coverage of each fuzzer during the 10 runs. The X-axis denotes the PUT, and the Y-axis is the number of covered paths.

[8, 23]. Therefore, we follow this configuration to conduct the evaluation. In addition, AFLFast, FairFuzz, and TortoiseFuzz are also following this configuration.

The initial seeds are taken from the LAVA-M dataset. We use the number of identified LAVA-M bugs as one of the metrics. We further present the execution speed to answer RQ3. All the evaluations last for 24 hours. We count the number of discovered bugs by the `lava_validation.py` script provided by Angora.

5.3.1 Identified Bugs

Table 12 shows the identified bugs of each fuzzer. ovAFLow outperforms QSYM, TIFF, AFLFast, FairFuzz, and TortoiseFuzz in all the PUTs. The reason is clear. Without customized mutation and seed prioritization strategies, QSYM, AFLFast, FairFuzz, and TortoiseFuzz cannot identify those memory corruption bugs. Additionally, TIFF suffers from performance overhead and triggers fewer bugs in a given time budget. Moreover, ovAFLow detects more bugs in base64, md5sum, and uniq than Angora and even exposes bugs not listed in the LAVA-M dataset. Through manual analysis, we find out that these unlisted bugs are memory corruption bugs. This proves that ovAFLow can identify hidden memory bugs.

5.3.2 Execution Speed

Fig. 7 is the speed of each fuzzer on the LAVA-M dataset. ovAFLow is slower than QSYM in three out of the four PUTs. This is similar to the evaluation results in the real-world programs. Most significantly, ovAFLow is about 10,000 times faster than TIFF. TIFF can only execute an input every few seconds, while ovAFLow can reach the speed of hundreds of executions per second. Our application of FTI rather than heavy-weight taint analysis results in this huge difference in speed. In addition, compared with AFLFast, FairFuzz, and TortoiseFuzz, the speed of ovAFLow is at the same level. This part of the evaluation answers RQ3 that ovAFLow is running with acceptable performance overhead.

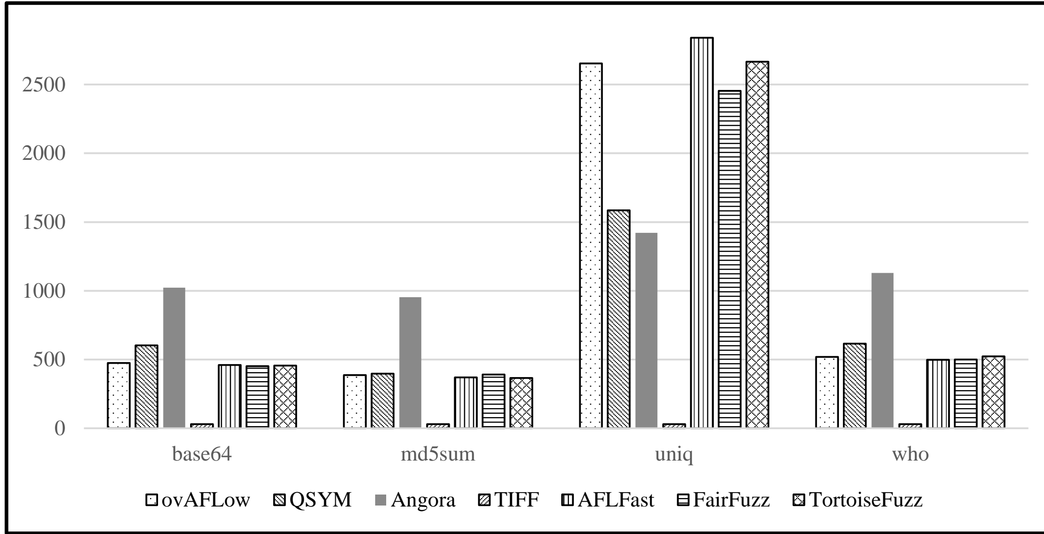


Fig.7. Execution speed of each fuzzer. The X-axis denotes the PUT, and the Y-axis is the number of executions per second.

Table 13. Average Path Coverage of Each Fuzzer in the LAVA-M Dataset

Fuzzer	Path Coverage
ovAFLow	226.8
QSYM	230(-1.4%)
Angora	248.5(-8.8%)
TIFF	131.3(+72.8%)
AFLFast	214.8(+5.6%)
FairFuzz	220.2(+3.0%)
TortoiseFuzz	203.8(+11.3%)

Note: The numbers in the brackets show the increase or decrease compared with the baseline fuzzers.

5.3.3 Path Coverage

Furthermore, we record the program paths each fuzzer covers in Fig. 8. We get similar results as in the real-world programs that ovAFLow covers more paths than TIFF in the four PUTs. Angora can outperform ovAFLow in three out of the four programs because of its constraint solving ability. In addition, ovAFLow outperforms AFLFast, FairFuzz, and TortoiseFuzz in most of the PUTs.

Table 13 shows the average covered paths of each fuzzer. As we can see from the table, the program paths of ovAFLow are 1.4% and 8.8% less than QSYM and Angora, respectively. ovAFLow outperforms TIFF, AFLFast, FairFuzz, and TortoiseFuzz by 72.8%, 5.6%, 3.0%, and 11.3%, respectively. In conclusion, though we are not aiming to increase program coverage, we still get better results than most of the baseline fuzzers.

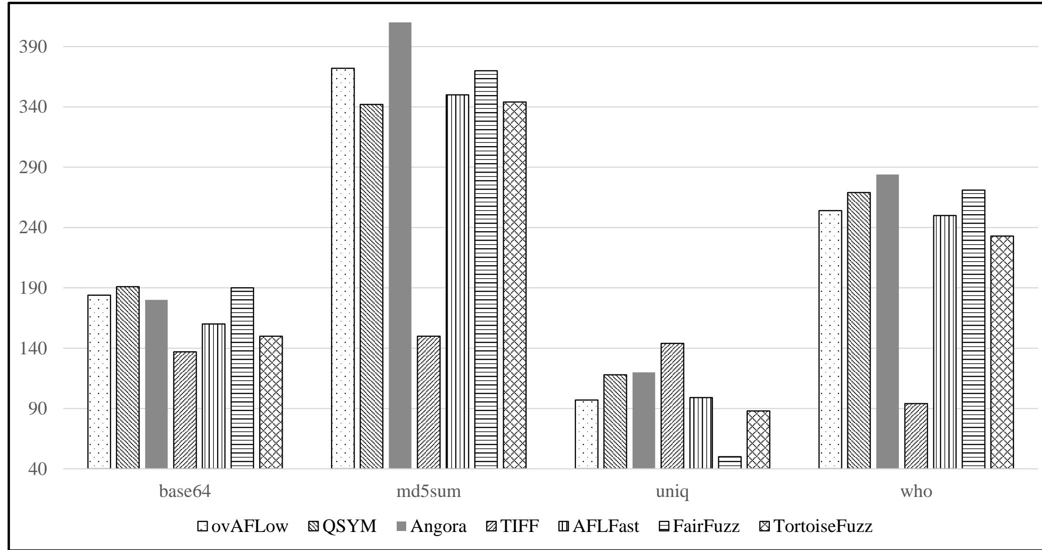


Fig.8. Path coverage of each fuzzer. The X-axis denotes the PUT, and the Y-axis is the number of covered paths.

5.4 Additional Evaluation Results

Table 14. New Bugs Discovered by ovAFLow

Target	Bug Description
mp42aac	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4HvccAtom.cpp:282:24 in AP4_HvccAtom ::AP4_HvccAtom(unsigned int, unsigned char const*)
mp42aac	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4AvccAtom.cpp:165:31 in AP4_AvccAtom ::AP4_AvccAtom(unsigned int, unsigned char const*)
mp42aac	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4Utils.cpp:548 AP4_BitReader ::SkipBits(unsigned int)
mp42aac	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4Dec3Atom.cpp:97 AP4_Dec3Atom:: AP4_Dec3Atom (unsigned int, unsigned char const*)
mp4tag	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4AvccAtom.cpp:88 AP4_AvccAtom ::Create(unsigned int, AP4_ByteStream&)
mp4tag	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4RtpAtom.cpp:51 AP4_RtpAtom:: AP4_RtpAtom(unsigned int, AP4_ByteStream&)
mp4tag	heap-buffer-overflow Bento4-1.6.0-637/Source/C++ /Core/Ap4AvccAtom.cpp:165 AP4_AvccAtom:: AP4_AvccAtom(unsigned int, unsigned char const*)
tiff2pdf	heap-buffer-overflow (tiff-4.1.0/build-orig-asan/mybin/bin/tiff2pdf+0x459d44) in __interceptor _memcpy.part.42
tiff2ps	heap-buffer-overflow tiff-4.1.0/tools/tiff2ps.c:2479:20 in PSDataColorContig
infotocap	heap-buffer-overflow ncurses/tinfo/captoinfo.c:644 _nc_infotocap
infotocap	stack-buffer-overflow ncurses/progs/dump_entry.c:1144 fmt_entry
infotocap	global-buffer-overflow (ncurses-6.2/build-orig-asan/mybin/bin/infotocap+0x460e85)

5.4.1 *New Bugs*

During the evaluation, we identify 12 new bugs which are listed in Table 14. All of them were reported to maintainers, and some were confirmed then fixed to the time of writing this paper. As we can see, all of them are overflow bugs, which are a subset of memory corruption bugs. These bugs are harmful, and they can lead to severe consequences such as denial-of-service. Our strategies to mutate the memory operation function arguments and memory access loop counts drive these overflow bugs to be triggered.

In addition, by mutating the target variables to extreme values, we can detect other types of memory bugs. For example, by mutating the `size` argument in `malloc`(), we identify several memory consumption bugs. These bugs can consume too much memory and make the operating system reboot. However, these identified bugs are already discovered by others. Therefore, they are not listed in the table.

Table 15. CVEs Discovered by ovAFLow

CVE ID	Description
2020-21064	A buffer-overflow vulnerability in the AP4_RtpAtom::AP4_RtpAtom function in Ap4RtpAtom.cpp of Bento4 1.5.1.0 allows attackers to cause a denial of service.
2020-21066	An issue was discovered in Bento4 v1.5.1.0. There is a heap-buffer-overflow in AP4_Dec3Atom::AP4_Dec3Atom at Ap4Dec3Atom.cpp, leading to a denial of service (program crash), as demonstrated by mp42aac.

Moreover, two CVEs based on these new bugs are assigned. We list them in Table 15. These CVEs demonstrate that these bugs are harmful, and they can cause security issues.

Table 16. Identified Memory Operation Functions and their Frequencies

Function Name	Frequencies
memcpy	11819
memset	4932
CRYPTO_free	3466
snprintf	1034
CRYPTO_malloc	895
memmove	735
malloc	701
BIO_printf	668
fprintf	561
archive_read_data	535
strncmp	430
CRYPTO_clear_free	317
archive_read_open_filename	237
strchr	226
BIO_snprintf	221
ft_mem_realloc	196
realloc	189
archive_write_data	161
strtol	140
strncpy	140
archive_write_open_memory	134
BIO_read	130
sprintf	129
fwrite	126
__archive_write_output	124
archive_read_open_memory	121
read_pbm_integer	112
ft_mem_alloc	103
strncat	84
strtoull	82
strtoll	81
open	81
CRYPTO_memcmp	76
app_malloc	75
write	66
read	59
strrchr	54
memchr	52
strtoul	41
fstat	33
vsnprintf	32
fgets	29
fseek	23
fputc	22

5.4.2 Memory Operation Functions

We use static analysis to identify memory operation functions in the Google fuzzer test suite. Table 16 shows the function names and the number of times they are identified as memory operation functions. Commonly used library functions are listed in the table, such as `memcpy`() and `memset`(). Meanwhile, we also collect other functions with high frequencies that may be missed with manual analysis, such as `CRYPTO_malloc`(). In total, we automatically identify 44 functions, and their arguments are used as taint targets.

5.5 Discussion

In Algorithm 1, we use offline statistics to identify the memory operation functions. However, online algorithms may also be effective in identifying the functions. We consider the trade-off between offline and online statistics. Fuzzing is sensitive to execution speed. Online algorithms possibly require complex program analysis. We think online algorithms may slow down the speed of fuzzing. Therefore, we use offline statistics, which are easy and direct.

6 Related Work

6.1 Seed Selection

In the fuzzing process, the fuzzer needs to choose a seed at the end of the previous round of fuzzing. It is important to select the best seed based on the goal of the fuzzer. When a seed is marked as favored, it will be selected with a higher probability in the following rounds. In addition, MemLock [26] and UAFL [27] choose seeds with more memory consumption and more UAF (use-after-free) sequences, respectively. Furthermore, AFLGo [28] and CollAFL [11] also select seeds with their specific goals. However, they require complex program analysis to finish the task. Unlike them, ovAFLow does not need additional static analysis to select the seeds. The process of FTI and the following seed selection are integrated into the original procedure of AFL, which require no complex operation.

6.2 Memory Bug Guided Fuzzing

Memory corruption is non-trivial in software, and fuzzing memory bugs has drawn the attention of researchers. TIFF [9] uses DTA to identify input bytes that can taint commonly seen memory operation functions. The building block of TIFF is a type-aware mutation strategy to efficiently trigger memory corruption bugs. In the beginning, TIFF manually collects library functions, such as `memcpy()`, and marks the arguments of these functions as target variables. Then, TIFF recognizes important input bytes with a heavy-weight analysis technique: dynamic taint analysis. Next, TIFF mutates the recognized taint input bytes to extreme values in the fuzzing process to trigger memory corruption. However, the memory operation functions are manually collected in TIFF, including only 17 functions. In contrast, ovAFLow automatically identifies 44 functions where memory bugs may happen. Meanwhile, the heavy-weight DTA in TIFF cannot fit the fuzzing process. We use light-weight detecting techniques in ovAFLow, such as FTI, to keep the fuzzing process fast.

Whereas for MemFuzz [10] and CollAFL [11], they share similar ideas that they identify memory accesses in each program input, and inputs with more memory accesses can be executed with a higher possibility in the following fuzzing campaign. The intuition behind this idea is to increase the chance of detecting memory bugs by performing more memory operations. However, their prioritization strategies are not efficient. The prioritization strategy is coarse-grained and cannot distinguish seeds with different numbers of taint bytes. To solve these problems, we use more precise taint-based seed prioritization strategies in ovAFLow to detect more memory bugs.

6.3 Fuzzing with FTI

FTI is a newly proposed technique in fuzzing. It is designed in replacement of heavy-weight taint analysis, such as DTA. SLF [19] adopts random mutations to mark inputs, inferring taints directly related to inputs. ProFuzzer [18] monitors the changes in control flow and partially infers the types of bytes. Our proposed FTI is different from these tools. The goal of SLF and ProFuzzer is to identify the accurate type of the input bytes. For example, they want to determine whether byte A of the input was working as an enumeration variable in the PUT. In addition, both of them mutate a byte 256 (2^8) times in the FTI process. This causes considerable performance overhead in identifying the taint input bytes. In contrast, the FTI of ovAFLow is integrated into the "bitflip-1" mutation of AFL. The "bitflip-1" process only mutates a byte eight times. The FTI in ovAFLow will not cause additional performance overhead.

GREYONE [17] performs complete byte-level mutation and monitors the changes to infer taint attributes. In ovAFLow, we implement our own FTI from scratch, which is different from GREYONE. Our taint targets are automatically collected from real-world programs and the PUTs. Furthermore, we integrate the FTI process into the "bitflip-1" stage of AFL, and it causes negligible performance overhead. In addition, the source code of GREYONE is unavailable. We will make our code public to boost the research in this field.

7 Conclusion

In this paper, we introduced a new memory bug guided fuzzer, ovAFLow. We broadened the vulnerable targets to memory operation function arguments and memory access loop counts. Furthermore, we used FTI to replace heavy-weight DTA in fuzzing. In eval-

uation, ovAFLow outperforms other fuzzers in bug detection and other aspects. We had the following discoveries. First, speed is the first priority in fuzzing, and we should never slow down the fuzzing process when adopting new techniques. Heavy-weight techniques which slow down the execution speed will have a negative effect on fuzzing. Second, the essence of fuzz testing is to detect more bugs. Our evaluation demonstrated that fuzzers should focus more on vulnerability detection ability, along with code coverage. Third, ovAFLow identifies many vulnerable targets that can be easily triggered into memory corruption, e.g., the arguments of memory operation functions. Programmers and researchers should spend effort in protecting these targets from being triggered into memory bugs. Additionally, in the future, we would like to add more precise program analysis techniques to ovAFLow. With these techniques, ovAFLow can generate more precise results and find memory corruption with a higher possibility.

References

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990, 33(12): 32-44. DOI: 10.1145/96267.96279.

[2] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017, 45(5): 489-506. DOI: 10.1109/TSE.2017.2785841.

[3] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware Evolutionary Fuzzing. In *Proc. the 2017 Network and Distributed System Security Symposium*, Feb. 2017, pp.1-14. DOI: 10.14722/NDSS.2017.23404.

[4] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May. 2018, pp.711-725. DOI: 10.1109/SP.2018.00046.

[5] Yun I, Lee S, Xu M, Jang Y, Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. the 2018 Usenix Security Symposium*, Aug. 2018, pp.745-761. DOI: 10.5555/3277203.3277260.

[6] Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proc. the 2018 ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2018, pp.475-485. DOI: 10.1145/3238147.3238176.

[7] Li Y, Ji S, Lv C, Chen Y, Chen J, Gu Q, Wu, C. V-Fuzz: Vulnerability-oriented evolutionary fuzzing. arXiv:1901.01142, 2019. https://arxiv.org/abs/1901.01142, Jan. 2019.

[8] Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, Su P. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Proc. the 2020 Network and Distributed System Security Symposium*, Feb. 2020, pp.1-17. DOI: 10.14722/ndss.2020.24422.

[9] Jain V, Rawat S, Giuffrida C, Bos H. TIFF: Using input type inference to improve fuzzing. In *Proc. the 2018 Annual Computer Security Applications Conference*, Dec. 2018, pp.505-517. DOI: 10.1145/3274694.3274746.

[10] Coppik N, Schwahn O, Suri N. MemFuzz: Using memory accesses to guide fuzzing. In *Proc. the 2019 IEEE Conference on Software Testing, Validation and Verification*, Apr. 2019, pp.48-58. DOI: 10.1109/ICST.2019.00015.

[11] Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z. CollAFL: Path sensitive fuzzing. In *Proc. the 2018*

IEEE Symposium on Security and Privacy, May. 2018, pp.679-696. DOI: 10.1109/SP.2018.00040.

[12] Zhou C, Wang M, Liang J, Liu Z, Jiang Y. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proc. the 2020 IEEE/ACM International Conference on Automated Software Engineering*, Dec. 2020, pp.858-870. DOI: 10.1145/3324884.3416572.

[13] Nagy S, Hicks M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proc. the 2019 IEEE Symposium on Security and Privacy*, May. 2019, pp.787-802. DOI: 10.1109/SP.2019.00069.

[14] Zhang C, Dong W Y, Ren Y Z. INSTRCR: Lightweight instrumentation optimization based on coverage-guided fuzz testing. In *Proc. the 2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology*, Aug. 2019, pp.74-78. DOI: 10.1109/CCET48361.2019.8989335.

[15] Jia X, Zhang C, Su P, Yang Y, Huang H, Feng D. Towards efficient heap overflow discovery. In *Proc. the 2017 Usenix Security Symposium*, Aug. 2017, pp.989-1006. DOI: 10.5555/3241189.3241267.

[16] Qin F, Lu S, Zhou, Y. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proc. the 2005 International Symposium on High-Performance Computer Architecture*, Feb. 2005, pp.291-302). DOI: 10.1109/HPCA.2005.29.

[17] Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z. GREYONE: Data flow sensitive fuzzing. In *Proc. the 2020 Usenix Security Symposium*, Aug. 2020, pp.2577-2594. DOI: 10.5555/3489212.3489357.

[18] You W, Wang X, Ma S, Huang J, Zhang X, Wang X, Liang B. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proc. the 2019 IEEE Symposium on Security and Privacy*, May. 2019, pp.769-786. DOI: 10.1109/SP.2019.00057.

[19] You W, Liu X, Ma S, Perry D, Zhang X, Liang B. SLF: Fuzzing without valid seed inputs. In *Proc. the 2019 IEEE/ACM International Conference on Software Engineering*, May. 2019, pp.712-723. DOI: 10.1109/ICSE.2019.00080.

[20] Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, Whelan R. LAVA: Large-scale automated vulnerability addition. In *Proc. the 2016 IEEE Symposium on Security and Privacy*, May. 2016, pp.110-121. DOI: 10.1109/SP.2016.15.

[21] Aho A V, Sethi R, Ullman J D. Compilers, principles, techniques. Addison wesley, 1986.

[22] Zhang G, Zhou X, Luo Y, Wu X, Min E. PTfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 2018, 6: 37302-37313. DOI: 10.1109/ACCESS.2018.2851237.

[23] Lyu C, Ji S, Zhang C, Li Y, Lee W H, Song Y, Beyah R. MOPT: Optimized mutation scheduling for fuzzers. In *Proc. the 2019 Usenix Security Symposium*, Aug. 2019, pp.1949-1966. DOI: 10.5555/3361338.3361473.

[24] Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multiarmed bandit. In *Proc. the 2020 Usenix Security Symposium*, Aug. 2020, pp.2307-2324. DOI: 10.5555/3489212.3489342.

[25] Serebryany K, Bruening D, Potapenko A, Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proc. the 2012 Usenix Security Symposium*, Aug. 2012, pp.309-318. DOI: 10.5555/2342821.2342849.

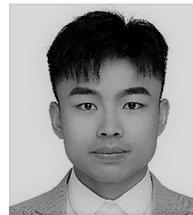
[26] Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, Liu T. MemLock: Memory usage guided fuzzing. In *Proc. the 2020 ACM/IEEE International Conference on Software Engineering*, Jun. 2020, pp.765-777. DOI: 10.1145/3377811.3380396.

[27] Wang H, Xie X, Li Y, Wen C, Li Y, Liu Y, Sui Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proc. the 2020 ACM/IEEE International Conference on Software Engineering*, Jun. 2020, pp.999-1010. DOI: 10.1145/3377811.3380386.

[28] Böhme M, Pham V T, Nguyen M D, Roychoudhury A. Directed greybox fuzzing. In *Proc. the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017, pp.2329-2344. DOI: 10.1145/3133956.3134020.

Gen Zhang received his B.S. and M.S. degrees in computer science and technology, in 2016 and 2018, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his Ph.D. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include fuzzing and software testing.

Peng-Fei Wang received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2011, 2013, and 2018, respectively, from the College of Computer, National University of Defense Technology, Changsha. His research interests include operating system and software testing.



Tai Yue received his B.S. and M.S. degrees in computer science and technology, in 2017 and 2019 from Nanjing University, Nanjing and the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his Ph.D. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include fuzzing and software testing.

Xiang-Dong Kong received his B.S. degree in computer science and technology in 2019, from the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his M.S. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include fuzzing.

Xu Zhou received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2007, 2009, and 2014, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and parallel computing.

Kai Lu received his B.S. degree and Ph.D. degree in 1995 and 1999, respectively, both in computer science and technology, from the College of Computer, National University of Defense Technology, Changsha. He is now a professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems, parallel computing, and security.