CSI-Fuzz: Full-Speed Edge Tracing Using Coverage Sensitive Instrumentation

Xiaogang Zhu, Xiaotao Feng, Xiaozhu Meng, Sheng Wen, Seyit Camtepe, Yang Xiang, *Fellow, IEEE*, and Kui Ren, *Fellow, IEEE*

Abstract—Coverage-guided fuzzing is one of the most effective solutions for vulnerability discovery. Among coverage-guided fuzzing, full-speed fuzzing, such as UnTracer, traces test cases only when they discover new coverage. Due to the high expense of tracing test cases, full-speed fuzzers improve the efficiency of fuzzing by tracing only coverage-increasing test cases. However, the existing full-speed fuzzer (i.e., UnTracer) is based on basic block coverage, suffering a severe problem called edge collision. Moreover, such fuzzers neglect the path frequency, which affects fuzzing effectiveness. In this article, we propose CSI-Fuzz, a fuzzer utilizing coverage sensitive instrumentation to address the problems of existing full-speed fuzzing. CSI-Fuzz directly instruments at edges, which solves the problem of edge collision. Meanwhile, CSI-Fuzz sets path identifiers to count the frequency of covered paths. Our CSI-Fuzz can be recognized as an add-on and seamlessly applied to existing coverage-guided fuzzers. We accordingly implement CSI-Fuzz based on two widely-adopted fuzzers, AFL and AFLFast, to evaluate its performance. The experiments demonstrate that CSI-Fuzz discovers more edges than AFL, AFLFast, and UnTracer. Additionally, CSI-Fuzz exposes more bugs than the other fuzzers.

Index Terms—Full-speed fuzzing, edge tracing, edge collision, path frequency

1 Introduction

Fuzzing is an effective vulnerability assessment technique which randomly generates inputs, monitors the execution of the target program for exceptions such as crashes, and utilizes the results of execution to guide the generation of more inputs. The exceptions deep in target programs manifest potential vulnerabilities. Many of these vulnerabilities, like buffer overflow, can have serious security implications such as causing breakouts of malware around the world [2]. Generally, fuzzing is considered as black-, white- or grey-box depending on the level of awareness on the target program. Black-box fuzzing starts with no information about the internal structure of the program. White-box fuzzing utilizes the program analysis on the source code and grey-box fuzzing uses instrumentation (i.e., code inserted into a program to monitor its components) to collect information about the program.

Various fuzzers have been developed for effective bug analysis purpose [8], [9], [10], [11], [35], [36]. Among the existing fuzzing solutions, coverage-guided fuzzing has been widely recognized and deployed. One of the successful coverage-guided fuzzing examples is Google's OSS-Fuzz platform [13], which adopts libFuzzer [17] and AFL [36], to continuously test real-world applications. As reported in GoogleBlog [3], the OSS-Fuzz platform has found over 1,000 bugs in five months. Grey-box coverage-guided fuzzing (GCF) uses dynamic or static instrumentation to obtain the knowledge of an execution path that is exercised by an input. If an input discovers new coverage, fuzzing will retain the associated input as a seed. A seed is used to generate more inputs via mutating some bytes of the seed. Otherwise, the input is discarded, and new inputs are generated from existing seeds to test target programs. This iterative loop related to the seed mutation and the input retention provides the chance to explore execution paths in an efficient manner, and increases the likelihood to disclose more vulnerabilities [8].

However, as analysed in UnTracer [22], such iterative loop introduces significant overhead because fuzzing traces all test cases. For each test case, fuzzing determines new coverage via comparing the current coverage with all discovered coverage. Because most test cases generated by fuzzing cannot discover new coverage, it will save much time if fuzzing skips tracing such test cases. Therefore, to increase the execution speed of fuzzing, UnTracer proposes the full-speed fuzzing which traces test cases only when fuzzing examines new basic blocks. However, the block-based fuzzing introduces a severe problem called edge collision, which occurs when two or more different edges in a control flow graph (CFG) are regarded as the same one. Specifically, new CFG edges may not necessarily indicate new basic blocks because edges provide fuzzing with the information of the connections between different blocks. If the block-based fuzzing does not have the information of connections, it cannot distinguish the same

- *Xiaogang Zhu, Xiaotao Feng, Sheng Wen, and Yang Xiang are with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia. E-mail: {xiaogangzhu, xfeng, swen, yxiang}@swin.edu.au.*
- *Seyit Camtepe is with DATA61 CSIRO, Sydney, NSW 2122, Australia. E-mail: Seyit.Camtepe@data61.csiro.au.*
- *Xiaozhu Meng is with Rice University, Houston, TX 77005 USA. E-mail: xiaozhu.meng@rice.edu.*
- *Kui Ren is with the University at Buffalo, Buffalo, NY 14203 USA. E-mail: kuiren@buffalo.edu.*

Manuscript received 17 Dec. 2019; revised 26 June 2020; accepted 9 July 2020. Date of publication 13 July 2020; date of current version 14 Mar. 2022. (Corresponding author: Xiaogang Zhu.) Digital Object Identifier no. 10.1109/TDSC.2020.3008826

basic block executed with different edges. Edge collision decreases the efficiency and effectiveness of fuzzing. First, the collision blurs fuzzing strategies, such as seed selection and mutation scheme, which impacts the efficiency of vulnerability exposure. Second, the collision could cause fuzzing to skip tests on conflicted paths, which will miss vulnerabilities in such paths [11].

Moreover, existing full-speed fuzzing cannot count path frequency accurately because it does not trace non-coverage-increasing test cases (i.e., test cases that do not discover new coverage). As path frequency is critical for fuzzers whose mutation schemes are based on the frequency of paths, existing full-speed fuzzing is not able to support such fuzzers. For example, AFLFast [8] calculates path identifiers based on current coverage, and uses the identifiers to count the frequency of paths. Then, it assigns more mutation chances for the low frequent paths. However, because existing full-speed fuzzers do not trace test cases that are non-coverage-increasing, the frequencies of associated paths cannot be counted. In this case, AFLFast wrongly assigns mutation chances to paths, and thus decreases the efficiency of fuzzing.

To solve the problems of existing full-speed fuzzing, a possible solution is to instrument directly at edges. However, this introduces new problems such as indirect control flow because it is hard to determine the targets of them statically [4], [20]. Meanwhile, Andriesse *et al.* [4] show that, in 981 real-world compiler-generated binaries, the missing target functions of indirect calls can be more than 20 percent of all functions. The numerous indirect calls lead fuzzing to missing some edges, which decreases the possibility to expose bugs. In order to instrument directly at edges, the problem of indirect edges has to be solved.

In this paper, we propose CSI-Fuzz, which utilizes coverage sensitive instrumentation to solve the problems of full-speed fuzzing. The instrumentation at edges will stop the execution of target binaries when fuzzing examines a new edge. Then, the next time fuzzing examines the same edge, the target program will continue to run. CSI-Fuzz instruments two types of edges, which are pre-determined edges and indirect edges. First, CSI-Fuzz directly instruments pre-determined edges, which are the edges that can be determined when statically analysing binaries. The instrumentation at pre-determined edges will be removed once these edges have been examined. For example, it directly instruments at condition-taken edges (e.g., the '`if`' branch in '`if (a==1){...} else{...}`' when '`a=1`') and condition-not-taken edges (e.g., the '`else`' branch in the previous example). Second, CSI-Fuzz instruments at the source blocks of indirect edges, including jump sites (e.g., switch statements) and call sites (e.g., virtual function calls), of which the instrumentation differentiates edges at runtime. Therefore, CSI-Fuzz can trace test cases only when they discover new edges, rather than basic blocks. Meanwhile, CSI-Fuzz sets path identifiers by calculating hash of path marks, which are the first new edges when new paths are found. Our solution to edge collision problem can improve the performance of other coverage-guided fuzzers because the coverage information is critical to those fuzzers. Note that, our solution is not to replace the existing fuzzers, but to collaborate with the existing ones.

Our CSI-Fuzz is built atop the modified coverage-guided fuzzer AFL, and uses Dyninst [21] to instrument target binaries. Besides, we also develop CSI-Fuzz(AFLFast), integrating CSI-Fuzz with another coverage-guided fuzzer, AFLFast. The evaluation demonstrates that CSI-Fuzz is more effective and efficient than AFL, AFLFast and Untracer in terms of edge discovery. For the bug discovery, CSI-Fuzz exposes more unique bugs than AFL, AFLFast and UnTracer. Moreover, CSI-Fuzz discloses a bug that other fuzzers are not able to find. In summary, the key contributions of this paper are three-fold:

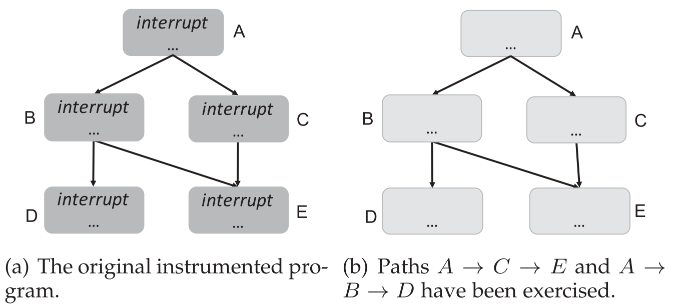
- We analyse the problems of existing full-speed fuzzing. First, it introduces problem of edge collision. Second, it cannot count path frequency accurately. Both of the two problems affect the efficiency and effectiveness of fuzzers.
- We accordingly design CSI-Fuzz utilizing coverage-sensitive instrumentation. It instruments directly at pre-determined edges to only trace coverage-increasing test cases. For the indirect edges, it instruments at the source blocks of them to conduct the full-speed fuzzing.
- We implement CSI-Fuzz based on coverage-guided fuzzers AFL and AFLFast. The results of experiments demonstrate that CSI-Fuzz is effective and efficient in terms of edge discovery, execution speed, and bug discovery.

We open-source our CSI-Fuzz on Github for further improvement. The code is available at https://github.com/Vul4Vendetta/csi-afl.

2 Overview of CSI-Fuzz

2.1 Background of Fuzzing

AFL is a widely used GCF and many existing fuzzers [7], [8], [18], [27], [36], [38] are developed based on AFL. As the primer of our idea, we introduce the fuzzing logic of AFL in this section. AFL tests a target program in an iterative fuzzing loop, which mainly includes seed selection, seed mutation, and coverage monitoring. It first sets some initial seeds, which are some inputs chosen by users. Then, it builds up a seed queue, which includes all the inputs that can discover new coverage. Based on the performance of each seed (e.g., the execution speed or the size of the seed), AFL selects the seed with the best performance in the queue to generate more inputs. Specifically, AFL mutates some bytes of the selected seed, and uses the result of mutation (i.e., a new input) to test the target program. After the new input has tested the target program, AFL checks whether the input discovers new coverage. The new coverage is indicated by new edges or different hit-counts of edges. AFL uses a bitmap to record the edge coverage, and each byte in the bitmap indicates the state of an edge. To determine new coverage, AFL traverses the bitmap and checks each byte. If the input discovers new coverage, it is retained as a new seed and added into the seed queue. Otherwise, the input is discarded. However, AFL traces every test case, which wastes much time because most test cases cannot discover new coverage. The mutation of a seed continues until fuzzing consumes up all the mutation energy, i.e., the times of mutating a seed. Then, AFL goes back to select a new winner seed and continues to generate more inputs.



interrupt
...
A
B
C
D
E
(a) The original instrumented program.
(b) Paths $A \to C \to E$ and $A \to B \to D$ have been exercised.

Fig. 1. Edge collision of block-based fuzzing. If paths $A \to C \to E$ and $A \to B \to D$ have been exercised, the interrupts at blocks are removed. Then, the new edge BE will be regarded as an old one, resulting in edge collision.

To improve the efficiency of the fuzzing loop, UnTracer skips tracing non-coverage-increasing test cases via inserting interrupt at the start of each uncovered basic block. Then, it removes the interrupts at the covered blocks, as shown in Fig. 1. At the beginning of each block, an interrupt is inserted to terminate the execution, as shown in Fig. 1a. When fuzzing has exercised the paths $A \to C \to E$ and $A \to B \to D$, the interrupts at the covered blocks A, B, C, D, E are removed, as shown in Fig. 1b. Therefore, UnTracer speeds up fuzzing via improving the coverage monitoring in the fuzzing loop. However, this block-based strategy introduces severe edge collision to fuzzing. As shown in Fig. 1, because paths $A \to C \to E$ and $A \to B \to D$ have been exercised, the interrupts at blocks A, B, C, D, E are removed. Therefore, the target program will continue to run when the path $A \to B \to E$ is exercised, indicating that the path includes no new edges. However, the edge BE is a new edge but is regarded as an old one, showing that edge collision occurs in this example.

2.2 Edge Instrumentation

CSI-Fuzz is a GCF, which statically instruments edges of target binaries so that it can conduct full-speed edge tracing. Fig. 2 shows the overview of CSI-Fuzz, which instruments the target binary using coverage sensitive instrumentation. The instrumentation at edges is sensitive to the change of coverage, and helps fuzzing monitor new edges without tracing all test cases. When a test case discovers a new edge, CSI-Fuzz traces the target binary and updates the information of examined edges. Therefore, CSI-Fuzz traces edges only when it examines a new edge. Besides, the new edge information (i.e., path marks) helps calculate path identifiers, which can count the frequency of covered paths.

CSI-Fuzz instruments at two types of edges which are pre-determined edges and indirect edges. The pre-determined edges include condition-taken edges and condition-not-taken edges. A condition-taken edge is an edge taken when a test case satisfies comparison instruction. Otherwise, it is a condition-not-taken edge. Other pre-determined edges are unconditional-jump (i.e., a jump without any branch) edges and no-jump edges (i.e., an edge without any jump). The other type is indirect edges including indirect calls and indirect jumps. When a test case examines a new edge, the execution of target binary will be terminated. Then, CSI-Fuzz removes the instrumentation at covered pre-determined edges so that the future executions will not be terminated when the same edges are examined. This solution improves the execution speed because new coverage is determined by the target binary itself, i.e., fuzzing skips tracing non-coverage-increasing test cases. Fig. 3 shows how CSI-Fuzz instruments at pre-determined and indirect edges. In Fig. 3b, the pre-determined edge AB is examined, thus the instrumentation at edge AB will be removed in the later executions.

The challenge is to instrument indirect edges because the targets of indirect edges are hard to get via statically analysing binaries. Therefore, instead of instrumenting indirect edges, CSI-Fuzz instruments at the indirect call sites and indirect jump sites, where we can accurately determine the indirect control flow targets at run time. When the target program reaches a source block of an indirect edge, it will compare the current target block to the historical ones. The historical target blocks are the ones that have been examined before. If the current target block exists in the historical ones, it indicates that the current indirect edge has been examined. Therefore, CSI-Fuzz continues the execution of

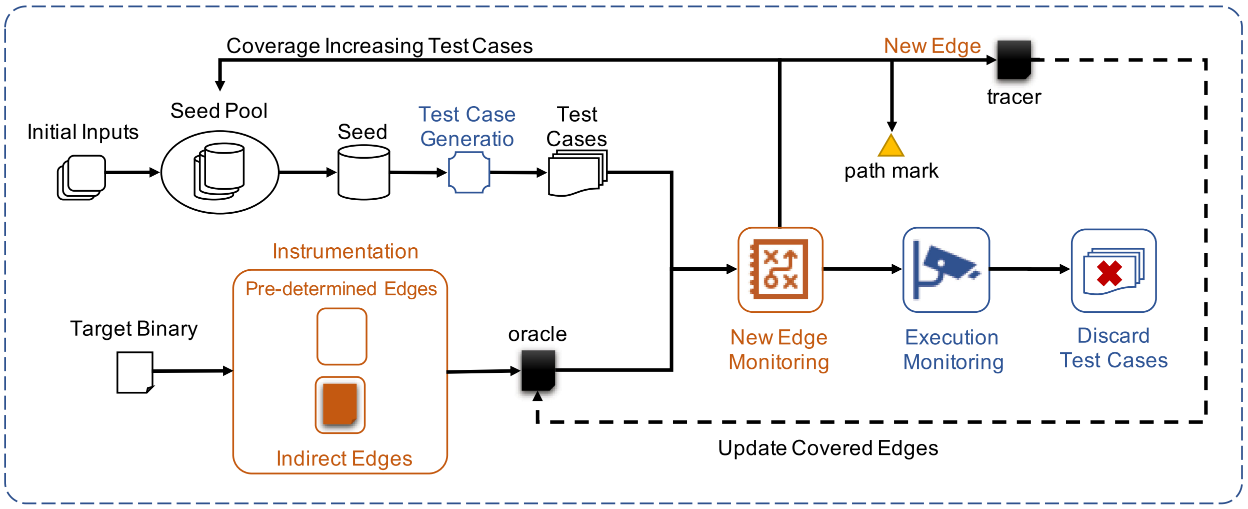
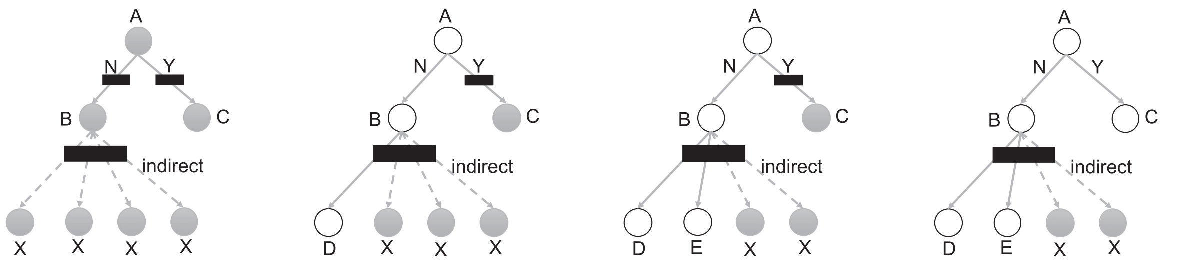


Fig. 2. Workflow of CSI-Fuzz. The yellow components indicate the coverage sensitive instrumentation. It instruments at two types of edges, pre-determined edges and indirect edges. When it discovers new edges, it will run the tracer to update covered edges and remove the instrumentation at covered edges in oracle.



(a) The original instrumented program. (b) Edges AB and BD are discovered. (c) Edges AB, BD and BE are discovered. (d) Edges AB, AC, BD and BE are discovered.

Fig. 3. Pre-determined edges and indirect edges. Instrumentation at pre-determined edges and indirect edges will stop the execution if new coverage is discovered. The black boxes are the instrumentation at edges. 'X' means the block is unknown. 'N' means the condition-not-taken edge while 'Y' means the condition-taken edge.

the target program. Otherwise, the current indirect edge is newly found, and the target program will then exit. In Figs. 3b and 3c, indirect edges BD and BE are newly discovered, the program will exit at those edges. Because edges BD and BE are recorded, the next time fuzzing will not terminate executions that examine edges BD and BE.

2.3 Full-Speed Edge Tracing

The full-speed edge tracing only traces test cases that discover new edges. To only trace coverage-increasing test cases, CSI-Fuzz instruments a target binary and generate two different instrumented binaries. One is called *oracle*, which terminates at new edges. The other is called *tracer*, which updates information of covered edges.

2.3.1 Oracle

The *oracle* is an instrumented binary utilized to conduct coverage-increasing edge tracing and it is invoked most of the time during fuzzing. Edges in *oracle* are instrumented with a function call to *exit*(). If a test case examines a new edge in the *oracle*, the *oracle* will exit. In this case, CSI-Fuzz cannot get all the edges along the path that the test case exercises. Therefore, CSI-Fuzz uses the *tracer* to get all the edges of that path and then removes instrumentation or records target blocks at those edges in *oracle*. Fig. 3 shows how *oracle* is instrumented.

2.3.2 Tracer

The *tracer* is an instrumented binary utilized to update the information of examined edges. CSI-Fuzz does not instrument *tracer* with *exit*() but with instructions recording all the edges along a path. The *tracer* provides the *oracle* with all edges along a path examined by the current test case. Then, CSI-Fuzz removes instrumentation or record target blocks at those covered edges in *oracle*. Note that, besides *oracle*, the *tracer* can also trigger a crash. Because *oracle* exits when examining a new edge, it may exit before triggering a crash. The *tracer*, when given the same input, would then trigger the crash that would happen after discovering the new edge.

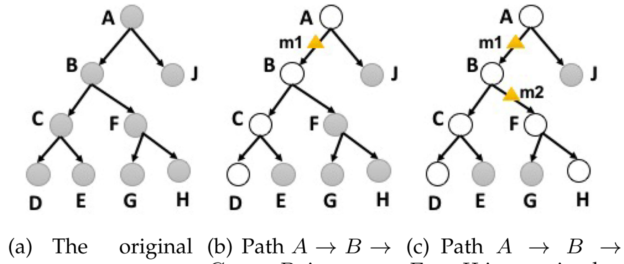
2.3.3 Crasher

We use a *crasher* to determine a new crash during fuzzing. The instrumentation of *crasher* is similar to the *oracle*, and will terminate execution when a new edge is examined. This solution saves time because one crash may be triggered many times. The *crasher*, which is instrumented *exit*() at uncovered edges, is invoked only when the *oracle* or *tracer* triggers a crash. Then, the test case triggering a crash in the *oracle* or *tracer* is utilized to examine the *crasher*. If the *crasher* terminates execution instead of crashing, it indicates that the test case triggers a new crash. Then the *crasher* removes instrumentation or records edges along the path examined by the test case. Otherwise, if the *crasher* crashes, it suggests that the test case triggers an old crash.

2.4 Path Identifier

For edge coverage-guided fuzzing, when a test case discovers a new edge in a target program, the test case is regarded as finding a new path. It is reasonable because a path consists of edges, and the existing paths do not contain the new edge. CSI-Fuzz follows this convention and leaves a mark on the new edge. Then, Fuzzing calculates path identifiers, which are used to record paths, based on all marks on the path. Using all marks on a path to calculate path identifiers enable us to distinguish paths that do not examine any new edge.

Fig. 4 shows the process of creating marks on paths. Fig. 4a is a target program which has not been examined and has all the instrumentation. If a first test case examines the path $A \rightarrow B \rightarrow C \rightarrow D$, the first new edge AB is set as a mark of the path $A \rightarrow B \rightarrow C \rightarrow D$, as shown in Fig. 4b. When the *oracle* meets the first new edge AB, it exits. CSI-Fuzz then invokes *tracer* to track all edges along the path, i.e., edges AB, BC and CD. Then, if a second test case examines the path $A \rightarrow B \rightarrow F \rightarrow H$, the first new edge BF is set as a mark of the path $A \rightarrow B \rightarrow F \rightarrow H$, as shown in Fig. 4c.



(a) The original program. (b) Path $A \rightarrow B \rightarrow C \rightarrow D$ is examined. (c) Path $A \rightarrow B \rightarrow F \rightarrow H$ is examined.

Fig. 4. Path identifiers. The yellow triangle is a path mark, which is the first new edge along a path.

Finally, CSI-Fuzz uses all the path marks along the same path to calculate the path identifier. For example, if another test case also examines the path $A \rightarrow B \rightarrow F \rightarrow H$ in Fig. 4c, CSI-Fuzz will go through two marks *m1* and *m2*. CSI-Fuzz uses the marks *m1* and *m2* to calculate the path identifier for path $A \rightarrow B \rightarrow F \rightarrow H$.

Algorithm 1. The Overall Algorithm of CSI-Fuzz

```
Input: $P$: the target program
Data: $e$: an edge
      $E$: a set of edges
      $t$: a test case generated by AFL
      $T$: the set of all coverage-increasing test cases
      $M$: the set of edges used as marks
1:  AFL_SETUP ()
2:  $E$ = getAllEdges ($P$)
3:  $P_O$ = InstOracle ($P, E$)
4:  $P_T$ = InstTracer ($P, E$)
5:  StartForkserver ($P_O, P_T$)
6:  while 1 do
7:      $t$ = chooseTestCase ()
8:      if isEdgeIncrease ($P_O, t$) then
9:          add first $e$ to $M$
10:         add $t$ to $T$
11:         $E_{trace}$ = traceEdges ($P_T, t$)
12: // Check loops
13:         if loopsExist () then
14:             markTestCase ($t$)
15:         end if
16:         stopForkserver ($P_O$)
17:         $P_O$ = removeInst ($P_O$, $E_{trace}$)
18:         addMarks ($P_O$, $M$)
19:         GETIndirectEdges () ▷
20: // Restart forkserver
21:         STARTForkserver ($P_O$)
22:     end if
23: end while
```

3 Implementation of CSI-Fuzz

CSI-Fuzz can be built atop different coverage-guided fuzzers, such as AFL [36] and AFLFast [8]. Currently we develop CSI-Fuzz(AFL) and CSI-Fuzz(AFLFast) based on AFL and AFLFast, respectively. For the convenience of usage, CSI-AFL is short for CSI-Fuzz(AFL) while CSI-AFLFast is short for CSI-Fuzz(AFLFast). CSI-Fuzz utilizes the instrumentation tool Dyninst to insert function *exit*(N) at edges. When a new edge is examined, the target program will exit with a special code. Then, CSI-Fuzz removes the instrumentation of *exit*(N) at pre-determined edges or records the targets of indirect edges. The next time when fuzzing examines the same edges, the target binary will not exit. CSI-Fuzz uses the exit code $N = 66$ for pre-determined edges and $N = 67$ for indirect edges. These two exit codes are chosen because they are different with the system ones.

Algorithm 1 shows how CSI-Fuzz conducts fuzzing. After the initial setup (line 1), CSI-Fuzz instruments *oracle* and *tracer* with different schemes (lines 2 - 4). The *oracle* terminates at new edges while *tracer* records all edges along the current execution path. Because *oracle* uses information from *tracer* to update instrumentation, CSI-Fuzz instruments them with the same edge information (e.g., edge identifiers). Then CSI-Fuzz infinitely test the target binary until user terminates fuzzing loop (line 6). In the infinite loop, if the *oracle* terminates because of exit code 66 or 67, the *oracle* examines a new edge. Then CSI-Fuzz sets the first new edge as a path mark, and saves the associated test case as a new seed (lines 8 - 10). After that, the *tracer* is invoked to trace all edges along the same execution path exercised by the current test case (line 11). With the edge information from *tracer*, CSI-Fuzz removes the instrumentation at the pre-determined edges that are traced by *tracer* (lines 16 - 17). Because CSI-Fuzz uses path identifiers to support fuzzers such as AFLFast, it adds path marks to *oracle* to calculate path identifiers (line 18). To share the information of indirect edges between *oracle* and *tracer*, the *oracle* updates the indirect edges before it starts the forkserver (lines 19 - 21). A forkserver forks a process to execute the target program so that fuzzing does not need to get indirect edges repeatedly, which saves time for fuzzing. As loops in programs are complicated, CSI-Fuzz gives more mutation energy to test cases that exercise loops (line 14). On the other hand, loops are neglected in UnTracer.

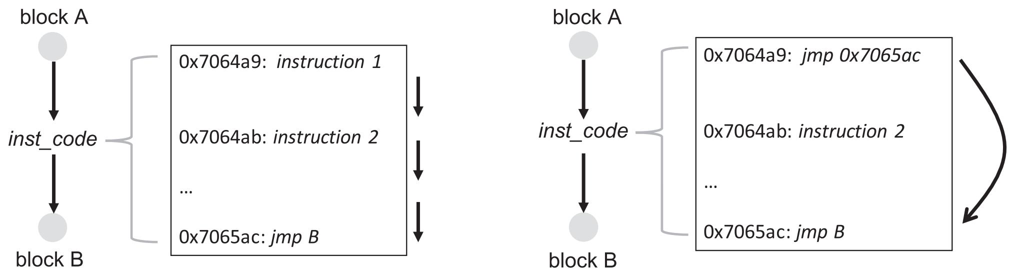
3.1 Edge Coverage

CSI-Fuzz statically instruments *oracle* and *crasher* to differentiate edges. When a new edge is examined, the instrumented program exits with exit code 66 or 67. Usually, when a program is executed successfully, the exit code is 0. Otherwise, when it has some errors, the exit code will be 1. Therefore, the special exit code 66 or 67 are not confused with system exit code. As shown in Algorithm 2, different types of edges are instrumented with different instructions.

Algorithm 2. Edge Instrumentation

```
Data: $b_t$: the target block of current edge
      $R_t$: the set of target blocks having been examined
1:  $R_t = \varnothing$
2:  if isPredeterminedEdge() then
3:      // $N$ is the exit code
4:      $exit(N)$
5:  else if isIndirectEdge() then
6:      $R_t$ = getExistingTargets()
7:      if $b_t$ not in $R_t$ then
8:          add $b_t$ to $R_t$
9:          $exit(N+1)$
10:     end if
11: end if
```

At the pre-determined edges that are not examined, CSI-Fuzz inserts *exit(66)* directly (lines 2 - 4). CSI-Fuzz first assigns edge identifiers to each pre-determined edge and stores the covered edge identifiers in shared memory. Therefore, when *tracer* records the current examined edges in the shared memory, *oracle* can quickly get the covered edges via looking up the shared memory. When a new edge is examined, CSI-Fuzz will remove the instrumentation at pre-determined edges. For indirect edges, CSI-Fuzz instruments at the call and jump sites of them. At a call or jump site, the instrumentation gets all the historical target blocks before they are compared with the current target block



(a) The original instrumented code. The inserted instructions are executed when edge AB is examined.

(b) Jumping over instrumentation. The first instruction is replaced by an unconditional jump.

Fig. 5. The removal of instrumentation. To remove the instrumentation at an edge, CSI-Fuzz replaces the first instruction of instrumentation with an unconditional jump, which jumps over the entire instrumentation at the edge.

(lines 6 - 7). If the current target block does not exist in the recorded ones, it is a new edge. Then, the current target block is added into the historical ones before exiting with exit code 67 (lines 8 - 9). For an indirect edge, the addresses of both the source block and the target block are recorded in a file. The list of historical indirect edges is read into the memory before forkserver. Therefore, the records of target blocks associated with each source block can be searched in the list.

3.2 Removal Scheme

We use Dyninst to instrument target binaries and Dyninst allows users to modify CFGs including edges [6]. In our first implementation, we improve Dyninst to support re-instrumentation, which instruments the instrumented target binaries (i.e., *oracle* or *crasher*) when new edges are examined. However, the re-instrumentation has to instrument target binaries from scratch, which has high overhead and wastes much time. We then realise that Dyninst inserts code snippets between blocks to accomplish instrumentation. Therefore, in our current implementation, we improve Dyninst to help jump over the inserted snippets, which is much faster than re-instrumentation. Specifically, we modify Dyninst to output a mapping from an edge in the target program to the address range of instrumentation for the edge. When CSI-Fuzz removes instrumentation for an edge, it looks up the instrumentation mapping to determine where the instrumentation is and replaces the first instruction of an inserted snippet with an unconditional jump, which jumps to the end of the snippet. Therefore, the instrumentation will not be executed even though the snippet still exists in the instrumented binary.

Fig. 5 exemplifies the removal of instrumentation at the edge AB. Before removal, the instructions in the code snippet are executed, as shown in Fig. 5a. Then, the first instruction is replaced with instruction *jmp 0x7065ac*, which jumps to the address *0x7065ac* unconditionally. Therefore, the code snippet is not executed when the edge AB is examined again. This scheme is faster than re-instrumentation because it removes instrumentation by replacing some bytes in the binary file.

3.3 Path Identifier

A queue is used in AFL to record information about the seeds, such as the name and the location of the seed file. CSI-Fuzz adds information of path identifiers into the queue so that it can count the frequency of execution paths. A path identifier is a hash value calculated based on path marks along the path. When a test case exercises an existing path, the *oracle* will write the current path marks into a shared memory. A path mark is the edge identifier of the first new edge in a path. Then CSI-Fuzz gets the bytes from the shared memory, and calculates hash values for path identifiers. Therefore, CSI-Fuzz knows which existing path is executed by comparing the current path identifier with the identifiers in the queue.

4 Evaluation of Binaries

To evaluate the performance of CSI-Fuzz, we run experiments to demonstrate the ability of CSI-Fuzz on edge discovery, execution speed, and bug discovery. We choose AFL [36], AFLFast [8] and UnTracer [22] to compare with our CSI-Fuzz. We choose AFL and AFLFast because both of them are edge-based fuzzing and they trace all test cases during fuzzing, which reduces the execution speed. On the other hand, we use UnTracer because it is a full-speed fuzzing but is based on basic blocks. Because the implementation of UnTracer[1] incurred segment fault at the time we ran our experiments, we re-wrote UnTracer based on our CSI-Fuzz, and implemented it carefully to keep its original functionality. We also develop a *tester* to count the edges that have been examined by fuzzers. Specifically, we use *tester* to instrument target binaries and use the seeds retained by each fuzzer to get the covered edges. In the experiments, we use 17 applications to evaluate the performance of fuzzing. For AFL and AFLFast, we evaluate them on binaries using two modes, QEMU mode and Dyninst mode. The QEMU mode is provided by AFL and AFLFast while the Dyninst mode is developed by us. We run the experiments on Linux 18.04 with *AMD Ryzen Threadripper 2990WX 32-Core Processor*.

4.1 Evaluation On QEMU Mode

AFL and AFLFast only provide QEMU, which is a simulator, to run binaries. Therefore, we use eight applications to evaluate the performance of fuzzing, and use the QEMU mode for AFL and AFLFast. We use AFL-QEMU and AFL-Fast-QEMU as the name of the QEMU mode of AFL and

1. https://github.com/FoRTE-Research/UnTracer-AFL

TABLE 1
The Performance of CSI-AFL and CSI-AFLFast in Edge Discovery Among 20 Trials

	CSI-AFL			CSI-AFLFast		
Application	+AFL-QEMU (%)	+UnTracer (%)	RSD (%)	+AFLFast-QEMU (%)	+UnTracer (%)	RSD (%)
base64	0.67	2.37	0.13	0.03	2.34	0.00
cjson	0.57	22.88	0.00	3.46	22.28	0.70
djpeg	1.46	4.84	0.83	6.74	4.90	5.22
md5sum	0.70	0.93	0.00	1.60	0.93	0.00
readelf	34.29	56.65	5.76	32.03	59.58	6.10
tcpdump	21.23	46.94	7.85	15.64	42.46	9.21
uniq	2.64	3.82	7.79	0.00	1.15	0.00
who	0.37	0.30	0.13	0.83	0.80	0.34

CSI-Fuzz discovers up to 34 percent more edges than AFL and AFLFast, and up to 60 percent more edges than UnTracer.

TABLE 2
Statistics of Applications for QEMU Mode, Including Pre-Determined Edges, Indirect Edges, and Basic Blocks

applications	version	#pre.	#indi.	#blocks
base64	LAVA-M	1,696	7	1,343
md5sum	LAVA-M	1,803	5	1,434
uniq	LAVA-M	1,731	8	1,369
who	LAVA-M	6,140	9	5,031
cjson	1.7.7	1,404	25	1,083
readelf	2.3	28,130	528	20,306
djpeg	libjpeg-9c	6,236	452	4,578
tcpdump	4.9.2	31,312	3,625	21,958

Note: *Instead of counting the indirect edges, we count the call sites of indirect edges.*

AFLFast, respectively. We run each fuzzer on an application for the same period of time, i.e., 24 hours, for 20 trials. For each experiment, we set 500 ms as the timeout, i.e., the current execution will terminate if it takes more than 500 ms. The eight popular applications from different fields are listed in Table 2. The third column of Table 2 is the number of pre-determined edges, the fourth column of Table 2 is the number of indirect edges, and the fifth column of the table is the number of basic blocks. For the eight applications, the number of blocks is less than the number of edges, indicating that edge collision may occur in block-based fuzzing.

To count the number of edges, we use *tester* to instrument target binaries and run the instrumented binaries with the seeds retained by each fuzzer. Table 1 shows the edge discovery of each fuzzer. The table shows that CSI-AFL discovers up to 34.29 percent more edges than AFL-QEMU and up to 56.65 percent more edges than UnTracer. Meanwhile, CSI-AFLFast finds up to 32.03 percent more edges than AFLFast-QEMU and up to 59.58 percent more edges than UnTracer. Besides, Tables 1 and 2 indicate that UnTracer has a severe problem of edge collision when fuzzing on programs such as `readelf` and `tcpdump`. On applications `readelf` and `tcpdump`, CSI-Fuzz discovers more than 40 percent more edges than UnTracer, while the numbers are less than 5 percent on other applications except `cjson`. On the other hand, for programs such as `base64`, the number of edges discovered by UnTracer tends to be close to CSI-Fuzz because such programs has a low possibility of edge collision for UnTracer.

To further research on the performance of edge discovery, we use the *tester* to count the number of condition-taken edges, condition-not-taken edges, indirect jumps, and indirect calls, which are shown in Fig. 6. On average, CSI-Fuzz discovers more edges than AFL and AFLFast. For example, CSI-AFL discovers 38.12, 32.85, 17.3, 23.42 percent more condition-taken edges, condition-not-taken edges, indirect jumps and indirect calls than AFL, respectively. Meanwhile, the numbers are 35.33, 29.59, 28.42 and 20.49 percent when comparing CSI-AFLFast to AFLFast.

To explain more details of edge discovery, Fig. 6 shows the results of different types of edges. In Fig. 6, the number of condition-not-taken edges is larger than the number of condition-taken edges, which implies that it is hard for fuzzing to generate test cases to solve path constraints.

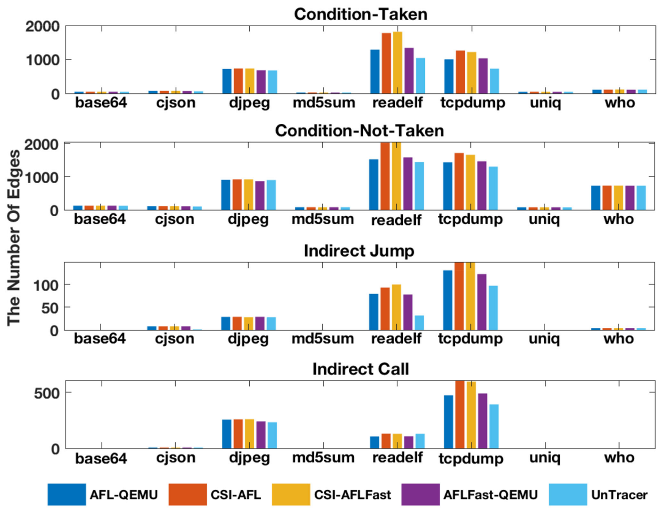


Fig. 6. The average number of examined edges. CSI-Fuzz discovers more edges than their raw versions.

Therefore, most test cases cannot satisfy conditions and go through the condition-not-taken edges. This is not surprising because fuzzing generates test cases almost randomly. In order to solve the path constraints in target programs, many fuzzers [14], [24], [26], [30], [34] are designed. Note that, our solution is not designed to replace the existing fuzzers, but to cooperate with them and improve the performance of fuzzing. As for indirect edges, our CSI-AFL and CSI-AFLFast discover more indirect jumps or calls than AFL, AFLFast or UnTracer, which gives a higher chance for CSI-Fuzz to find bugs in target programs. The result indicates that our solution is effective at differentiating indirect edges.

The variability of edge discovery shows the randomness of fuzzers. The relative standard deviation (RSD)[2] is a standardized measure of the variability of a frequency distribution. Table 1 shows the RSD of CSI-Fuzz in edge discovery among 20 trials. The RSD is calculated from the data that one fuzzer is evaluated on one application in 20 trials. It shows that CSI-Fuzz performs stably in terms of edge discovery and most RSDs in Table 1 are smaller than 10 percent. Based on Table 1, it can be concluded that when the number of covered edges is large, CSI-Fuzz performs much better than

2. https://en.wikipedia.org/wiki/Coefficient_of_variation

TABLE 3
Statistics of Applications for Dyninst Mode, Including Pre-Determined Edges, Indirect Edges, and Basic Blocks

applications	version	#pre.	#indi.	#blocks
objdump	2.28	100,357	2,307	74,394
exiv2	0.27.1	119,634	2,882	101,479
nasm	2.14	23,676	184	18,173
pdftohtml	0.22.5	71,296	1,911	61,241
bison	3.0.4	20,800	226	16,721
cflow	1.5	7,488	58	5,807
lou_translate (liblouis)	3.2.0	299	1	240
listswf (libming)	0.4.8	8,670	108	7,494
asn1Parser (libtasn1)	4.12	178	2	155

Note: *Instead of counting the indirect edges, we count the call sites of indirect edges.*

UnTracer. When the number of covered edges is small, the performance of fuzzers is close.

4.2 Evaluation On Dyninst Mode

The execution speed (i.e., average number of executions per second) is one of the critical factors for fuzzing because a higher execution speed saves time for fuzzing to execute more test cases. Due to the native low execution speed of QEMU mode, we develop AFL-Dyninst and AFLFast-Dyninst to evaluate on the execution speed of fuzzers. Because all the five fuzzers, i.e., CSI-AFL, CSI-AFLFast, AFL-Dyninst, AFLFast-Dyninst and UnTracer, are implemented based on Dyninst, the bias of the instrumentation tool is eliminated. We then use nine applications, which are from different fields and have different sizes, to evaluate the performance of fuzzers, especially the performance of execution speed. The sizes of the nine applications range from about 200 edges to about 120,000 edges. Table 3 lists the applications we use in the experiments. The meaning of columns in Table 3 is as the same as the columns in Table 2. For the nine applications, the number of blocks is less than the number of edges, indicating that edge collision may occur in block-based fuzzing. We run each fuzzer on an application for the same period of time, i.e., 6 hours, for 5 trials. For each experiment, we set 1000ms as the timeout.

The results of execution speed are shown in Fig. 7, where we remove top 10 percent and bottom 10 percent data to reveal the median tendency of execution speed [22]. The median tendency reduces the impact of system interference. For each application, we convert the average speed of each fuzzer to the *relative execution speed* with respect to the mean value of five fuzzers. Fig. 7 reveals that CSI-AFL and CSI-AFLFast executes binaries faster than AFL-Dyninst and AFLFast-Dyninst, respectively. Specifically, on average of the nine programs, the execution speed of CSI-AFL is $1.6\times$ higher than AFL-Dyninst. On applications *nasm* and *pdftohtml*, CSI-AFL executes binaries more than $2.5\times$ faster than AFL-Dyninst. On the other hand, CSI-AFLFast executes $1.4\times$ faster than AFLFast-Dyninst on average.

Fig. 7 indicates that the execution speed of fuzzers tends to be close to each other on small applications (e.g., *asn1Parser* and *lou_translate*). On the other hand, for large programs, CSI-AFL and CSI-AFLFast execute binaries much faster than their raw versions. The reason is that tracing small applications occupies a little time while large programs need much time to trace coverage. Note that, CSI-Fuzz instruments extra code to deal with indirect edges. Meanwhile, CSI-AFLFast spends extra time on determining the path frequency of current execution. Therefore, the execution speed of UnTracer is higher than CSI-AFL on four applications, and higher than CSI-AFLFast on seven applications. Although UnTracer conducts fuzzing faster than CSI-Fuzz, it discovers much fewer edges than CSI-Fuzz, which indicates that the block coverage has severe problem of edge collision.

Fig. 8 is the result of edge discovery, which shows the average number of edges of five trials. For all the nine applications, CSI-Fuzz discovers more edges than UnTracer except on the application *asn1Parser*. All the five fuzzers discover the same edges on *asn1Parser* because the application is a small one. CSI-AFLFast discovers 289.8 more edges than UnTracer on the application *exiv2*. On the other hand, CSI-AFL discovers 342.0 more edges than UnTracer on the application *objdump*. Besides, on six applications, CSI-Fuzz discovers edges faster than UnTracer. On the three small applications *asn1Parser*, *lou_translate* and *listswf*, CSI-Fuzz discovers edges as fast as the UnTracer does. Both Fig. 8 and Table 3 indicate that the block-based fuzzer UnTracer has the problem of edge collision.

4.3 Bug Discovery

We analyse the crashes discovered by each fuzzer on all the 17 applications listed in both Tables 2 and 3, and confirm bugs. Applications listed in Table 2 get their crash results from the evaluation on QEMU mode. Meanwhile, applications listed in Table 3 get their crash results from the evaluation on Dyninst mode. We use `afl-collect` from `afl-utils` [1] to remove the duplicated crashes and manually analyse the remaining crashes to confirm bugs. Table 4 shows the result of bug discovery during fuzzing. The column of *ave.* is the average number of unique bugs discovered during experiments. The column of *uniq.* is the number of unique bugs found by each fuzzer. Both CSI-AFL and

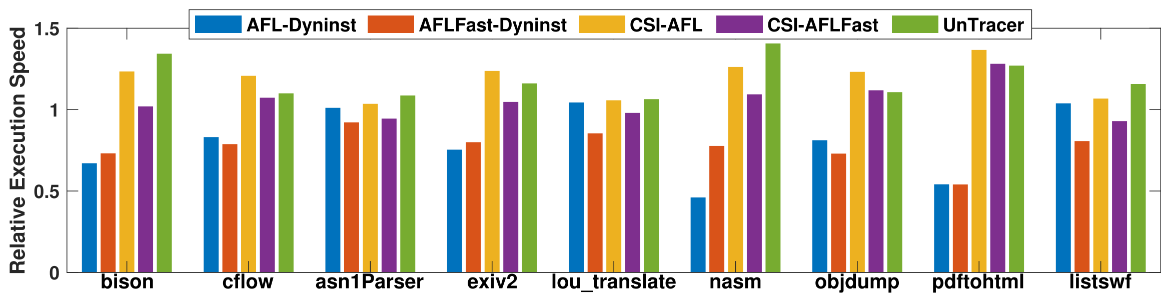


Fig. 7. The execution speed relative to the mean value. The execution speed of CSI-Fuzz is higher than their raw versions.

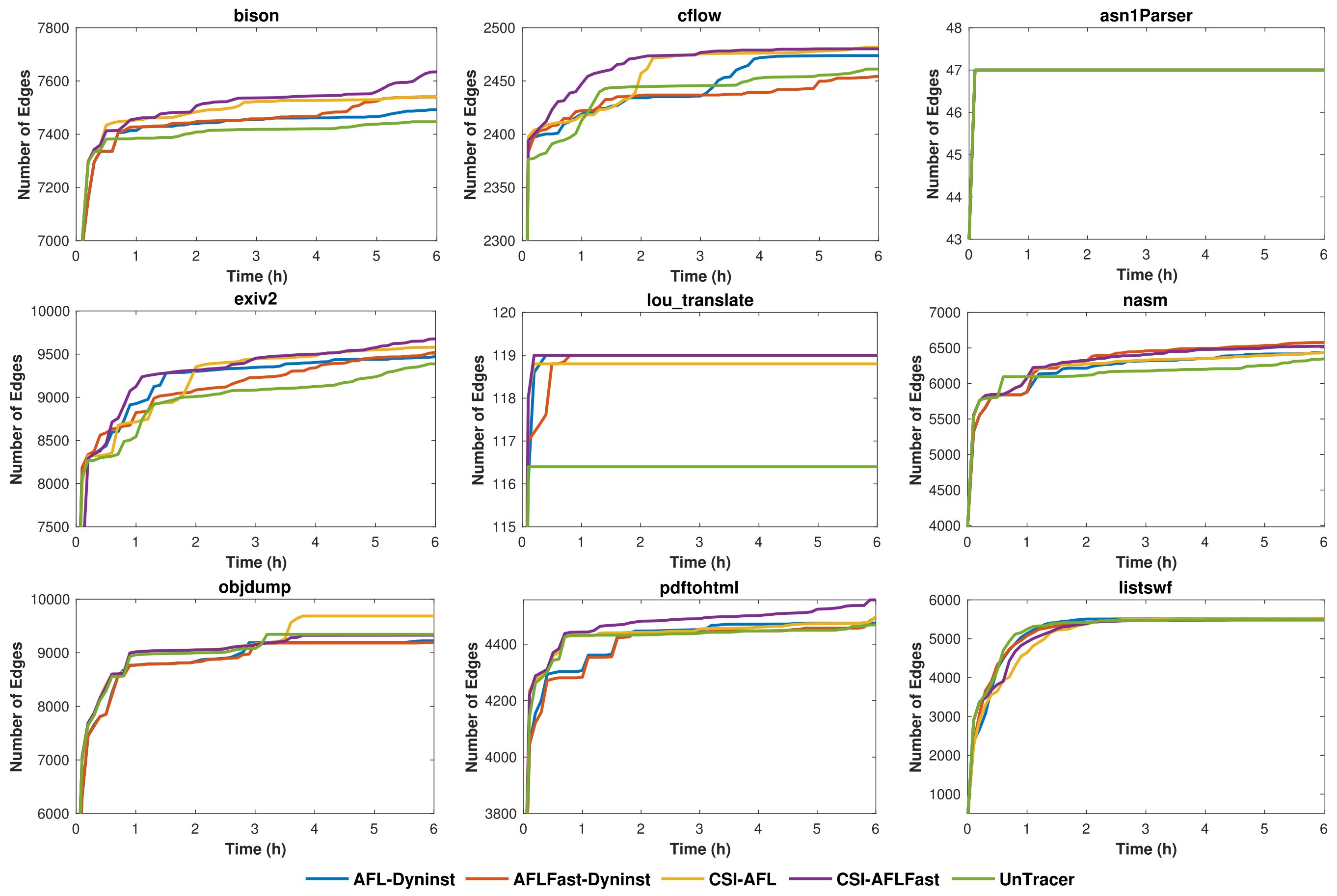


Fig. 8. The average number of edges discovered over time. CSI-Fuzz discovers more edges than other fuzzers.

CSI-AFLFast discover 6 unique bugs while other fuzzers only find 5 bugs. On average, CSI-Fuzz exposes the most bugs on each application, which indicates that CSI-Fuzz is more effective and efficient to expose bugs.

For the application *listswf*, CSI-Fuzz discovers a bug that is not discovered by any other fuzzer, and the crashing trace of the bug is shown in Fig. 9. This bug causes a segmentation fault when reading files. To trigger this bug, the input is required to contain two *block types*, which are *SWF_DEFINESPRITE* and *SWF_DOABC*, for the function *blockParse*. Meanwhile, these two *block types* call their own parsing functions (i.e., *parseSWF_DEFINESPRITE* and *parseSWF_DOABC*) in the exact order as shown in Fig. 9. Therefore, other fuzzers fail to expose this bug because of edge collision. Other fuzzers have the problem of edge collision so that they cannot retain the input that triggers the bug. For example, UnTracer cannot trigger the bug when the two functions *parseSWF_DEFINESPRITE* and *parseSWF_DOABC* have been examined separately. Therefore, the input that triggers the bug shown in Fig. 9 will be discarded because UnTracer does not discover new blocks in this scenario. However, because CSI-Fuzz is fast and do not have the problem of edge collision, it exposes the bug successfully.

5 Related Work

Coverage-guided fuzzing is one of the most successful fuzzing. It utilizes coverage information as feedback and guides fuzzing to generate effective test cases. AFL [36] utilizes bitmap to record coarse edge coverage, and traces all the test cases. Many fuzzers are developed based on AFL, but they are designed to resolve different challenges other than the

TABLE 4
The Number of Bugs Found by Fuzzers

Application	CSI-AFL		CSI-AFLFast		AFL		AFLFast		UnTracer	
	ave.	uniq.	ave.	uniq.	ave.	uniq.	ave.	uniq.	ave.	uniq.
who	1.0	1	1.0	1	1.0	1	1.0	1	0.7	1
readelf	2.0	2	2.0	2	1.4	2	1.35	2	2.0	2
lou_translate	1.0	1	1.0	1	1.0	1	1.0	1	1.0	1
listswf	1.6	2	1.4	2	1.0	1	0.8	1	1.0	1
total	5.6	6	5.4	6	4.4	5	4.15	5	4.7	5

Note: ave. *is the average number of unique bugs discovered by each fuzzer, and* uniq. *is the number of unique bugs found by each fuzzer. CSI-Fuzz discovers more bugs than others.*

Functions in a Crashing Trace	File & Line
main	main.c:350
readMovie	main.c:265
blockParse	blocktypes.c:145
parseSWF_DEFINESPRITE	parser.c:2316
blockParse	blocktypes.c:145
parseSWF_DOABC	parser.c:3481
parseABC_FILE	parser.c:3426
parseABC_CONSTANT_POOL	parser.c:3191
parseABC_NS_SET_INFO	parser.c:3083

Fig. 9. A bug in *listswf*. This bug is only found by CSI-Fuzz.

overhead of tracing test cases. AFLFast [8] improves the speed of path discovery but uses the same scheme to trace paths as AFL. MOPT-AFL [18] intends to improve the mutation strategy with Particle Swarm Optimization (PSO). However, MOPT-AFL only changes the mutation scheduling strategy in AFL. kAFL [27] is designed for the operating system (OS) kernel, which utilizes a hypervisor and Intel's Processor Trace (PT). The information from PT is provided as feedback to guide AFL. OSS-FUZZ [13] scales the AFL to large computing clusters. AFLGo [7] prioritizes seeds that have a shorter distance to the target vulnerable locations, integrating AFL via mainly replacing the seed selection scheme. CollAFL [11] is proposed to resolve the edge collision of AFL by using extended hash functions, which improves the performance of AFL.

Many fuzzers are designed to solve path constraints in target programs. Symbolic execution or concolic execution is effective in solving path constraints. The first fuzzing combined with concolic execution is proposed by Majumda and Sen [19]. Later, Driller, SAFL and DigFuzz are developed to improve the effectiveness of fuzzing. Driller [30] leverages concolic execution [28] to generate effective test cases for AFL and to reach more coverage. SAFL [33] uses symbolic execution to help fuzzing generate qualified initial seeds, which can exercise rare and deep paths. DigFuzz [37] first executes the target program utilizing AFL and then prioritizes difficult paths for concolic execution to process. However, Driller, SAFL and DigFuzz only integrate AFL with symbolic execution or concolic execution but do not pay attention to the problem of tracing all test cases. Pak [23] follows this idea of hybrid fuzzing, utilizing limited symbolic execution to find frontier nodes. QSYM [35] is a fast concolic execution engine, which helps the hybrid fuzzing scale to find bugs in complex software. However, symbolic execution struggles on the problem of path explosion, limiting the performance of hybrid fuzzing. Another technique to improve coverage of fuzzing is the taint analysis. Steelix [16], BuzzFuzz [12] uses taint tracing to locate the input bytes that influence errors in programs. Dowser [14] combines taint tracking and symbolic execution to find deep vulnerabilities in programs. Developed based on kAFL, REDQUEEN [5] optimizes fuzzing due to the observation that parts of the input directly correspond to the program state. In order to improve coverage, T-Fuzz [24] dynamically detects path constraints when AFL can no longer discover new coverage. Then these constraints are removed from the target program, allowing more coverage to be discovered.

Many fuzzers, such as AFL-GCC, CollAFL [11] and Angora, are implemented based on the information from source code. However, some fuzzers can run directly on binaries, namely binary-only fuzzers. If a fuzzer can run with target binaries, it can run with source code by merely compiling them. AFL-Dyinst [31] and AFL-Pin [32] instrument target binaries similarly to AFL-GCC with the same fuzzing logic. VUzzer [26] analyzes target binaries within static analysis tool IDA [15] and dynamic analysis tool Pin. T-Fuzz transforms part of the target binary using angr [29] and radare2 [25], which removes the sanity checks. TaintScope [34] utilizes Intel Pin to accomplish dynamic taint tracing on binaries. IDA, angr, radare2, Intel Pin and Dyninst are some of the common tools that are used to analyze target program binaries.

6 Discussion And Future Work

As Dyninst is an actively maintained tool, it may have bugs to conceal some crashes that are bugs. To solve this problem, we can improve Dyninst or use other instrumentation tools. Another issue is about the path identifiers. Our solution of path identifiers to differentiate paths still has some chances that two paths collide. However, other solutions to count the path frequency also causes path collision. For example, AFL calculates a hash value for the current bitmap and regard the hash value as the path identifier. This solution has the problem of path collision because the bitmap does not have the information of the order of edges. Note that, up to now, researchers have not proposed fuzzers based on path coverage due to the large overhead of tracing full path. In the future, we will research on a more effective method to set the path identifiers.

7 Conclusion

In this paper, we studied the problem called edge collision of full-speed fuzzing. The reason for edge collision is that existing full-speed fuzzers are based on block coverage, which lacks information of the order of blocks. Moreover, we discuss that path identifiers are critical because some fuzzing strategies are based on the path frequency. We accordingly propose CSI-Fuzz to resolve edge collision of full-speed fuzzing utilizing binary instrumentation. Binaries instrumented via CSI-Fuzz can be executed at high speed, allowing fuzzing to run more test cases. Then CSI-Fuzz sets path identifiers to improve fuzz effectiveness. Experiments show that this solution is effective and efficient at edge discovery and bug exposure. CSI-Fuzz finds more edges and bugs than AFL, AFLFast and UnTracer. As to the bug discovery, CSI-Fuzz exposes a bug that other fuzzers do not find because CSI-Fuzz executes binaries at high speed and solves the problem of edge collision. Moreover, CSI-Fuzz exposes more bugs during each run, implying the efficacy of CSI-Fuzz. The execution speed of CSI-Fuzz is faster than AFL and AFLFast. The execution speed of CSI-Fuzz is up to $2.5\times$ higher than AFL-Dyninst.

References

[1] afl-utils, Accessed: Jul. 2020. [Online]. Available: https://gitlab.com/rc0r/afl-utils

[2] Code red (computer worm), Accessed: Aug. 2019. [Online]. Available: https://en.wikipedia.org/wiki/Code_Red_(computer_worm)

[3] OSS-Fuzz: Five months later, and rewarding projects, Accessed: Aug. 2019. [Online]. Available: https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html

[4] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 583–600.

[5] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[6] A. R. Bernat and B. P. Miller, "Structured binary editing with a CFG transformation algebra," in *Proc. 19th Work. Conf. Reverse Eng.*, 2012, pp. 9–18.

[7] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

[9] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Security Privacy*, 2018, pp. 711–725.

[10] J. Corina *et al.*, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2123–2138.

[11] S. Gan *et al.*, "CollAFL: Path sensitive fuzzing," in *Proc. IEEE Symp. Security Privacy*, 2018, pp. 679–696.

[12] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.

[13] Google, "OSS-Fuzz - Continuous fuzzing of open source software," 2019. Accessed: Jun., 2019. [Online]. Available: https://github.com/google/oss-fuzz

[14] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. 22nd USENIX Conf. Secur. Symp.*, 2013, pp. 49–64.

[15] Hex-Rays, "IDA: Interactive disassembler," 2019. Accessed: Aug. 2019. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[16] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 627–637.

[17] LLVM, "libFuzzer–A library for coverage-guided fuzz testing," Accessed: Jun. 2019, 2019. [Online]. Available: https://releases.llvm.org/5.0.0/docs/LibFuzzer.html

[18] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, and Y. Song, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[19] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 416–426.

[20] X. Meng and B. P. Miller, "Binary code is not easy," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 24–35.

[21] B. Miller and J. Hollingsworth, "Dyninst: An API for program binary analysis and instrumentation," 2019. Accessed: Jun. 2019. [Online]. Available: https://dyninst.org/dyninst

[22] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 787–802.

[23] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," School of Comput. Sci. Carnegie Mellon Univ. 2012.

[24] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Security Privacy*, 2018, pp. 697–710.

[25] radare2 team, "radare2: Unix-like reverse engineering framework and commandline tools," 2019. Accessed: Aug. 2019. [Online]. Available: https://github.com/radare/radare2

[26] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.

[27] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 167–182.

[28] K. Sen, "Concolic testing," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 571–572.

[29] Y. Shoshitaishvili *et al.*, "SOK:(state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Security Privacy*, 2016, pp. 138–157.

[30] N. Stephens *et al.*,"Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2016, vol. 16, pp. 1–16.

[31] v. talos, "AFL-Dyninst," 2019. Accessed: Jun. 2019. [Online]. Available: https://github.com/talos-vulndev/afl-dyninst

[32] van Hauser, "AFL-Pin," 2019. Accessed: Jun. 2019. [Online]. Available: https://github.com/vanhauser-thc/afl-pin

[33] M. Wang *et al.*, "SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proc. 40th Int. Conf. Softw. Eng. Companion Proc.*, 2018, pp. 61–64.

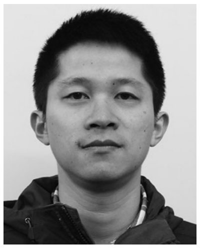
[34] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 497–512.

[35] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 745–761.

[36] M. Zalewski, "AFL," 2019. Accessed: Jun., 2019. [Online]. Available: https://github.com/mirrorer/afl

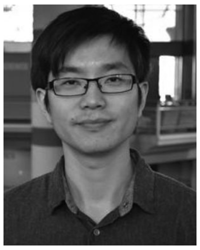
[37] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[38] X. Zhu *et al.*, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2019, pp. 658–663.



Xiaogang Zhu received the BEng degree from Xidian University, China, in 2012 and the MEng degree from Xi'an Jiaotong University, China, in 2015. He is working toward the PhD degree of computer science and engineering at the Swinburne University of Technology, Australia. Currently, his research is about searching vulnerabilities in programs. He is interested in detecting techniques such as fuzzing, machine learning, and symbolic execution.

Xiaotao Feng received the BEng degrees in computer science and engineering from South-West University, China, and Deakin University, Australia, in 2018. He is working toward the bachelor honours degree in computer science and engineering at the Swinburne University of Technology, Australia. He is working on the exploitation of security vulnerabilities at the Swinburne University of Technology, Australia. His research interests are programming languages, blockchain security, software testing, and embedded systems. His current project focuses on using fuzzing to detect vulnerabilities in embedded device firmware.



Xiaozhu Meng received the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, Hubei Province, China, and the master's and PhD degrees in computer science from the University of Wisconsin-Madison, Madison, Wisconsin. His research interest focuses on binary analysis and instrumentation for computer security and high performance computing. He is a research scientist at Rice University, Houston, Texas.

Sheng Wen received the PhD degree from Deakin University, Melbourne, Australia, in October 2014. Currently he has been working full-time as a senior lecturer with the Swinburne University of Technology, Australia. In the last six years, as an excellent early career researcher, he has published more than 50 high-quality papers in the last six years, including 35 journal articles (25 ERA A/A* journal papers and 11 IEEE Transactions journal papers) and 18 conference articles (top conferences like IEEE ICDCS) in the fields of information security, epidemic modelling and source identification. His representative research outcomes have been mainly published on top journals, such as the *IEEE Transactions on Computers* (TC), *IEEE Transactions on Parallel and Distributed Systems* (TPDS), *IEEE Transactions on Dependable and Secure Computing* (TDSC), *IEEE Transactions on Information Security and Forensics* (TIFS), and *IEEE Communication Survey and Tutorials* (CST).

Seyit Camtepe received the PhD degree in computer science from Rensselaer Polytechnic Institute, New York, in 2007. He is a senior research scientist at CSIRO Data61, Australia. From 2007 to 2013, he was with the Technische Universitaet Berlin, Germany, as a senior researcher and research group leader in security. From 2013 to 2017, he has worked as a lecturer with the Queensland University of Technology, Australia. His research interests include Pervasive security covering the topics autonomous security, malware detection and prevention, attack modelling, applied and malicious cryptography, smartphone security, IoT security, industrial control systems security, and wireless physical layer security.

Yang Xiang (Fellow, IEEE) received the PhD degree in computer science from Deakin University, Australia. He is the dean of Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the chief investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 200 research papers in many international journals and conferences, such as the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Information Security and Forensics*, and *IEEE Journal on Selected Areas in Communications*.

Kui Ren (Fellow, IEEE) received the BEng degree in chemical engineering, in 1998, the MEng degree in materials engineering, in 2001, both from Zhejiang University, China, and the PhD degree in electrical and computer engineering from Worcester Polytechnic Institute, Worcester, Massachusetts, in 2007. He is SUNY Empire Innovation professor and the director of the Ubiquitous Security and Privacy Research Laboratory (UbiSeC) with the Department of Computer Science and Engineering, University at Buffalo, State University of New York, where he joined, in 2012 as an associate professor and was promoted to full professor, in 2016. Previously, he has been with the Department of Electrical and Computer Engineering, Illinois Institute of Technology (IIT), Chicago, Illinois, where he received early tenure and promotion in five years starting 2007. His current research interests include data and computation outsourcing security in the context of cloud computing, wireless systems security in the context of Internet of Things, and crowdsourcing-based large-scale data acquisition. He has published frequently in peer-reviewed journal and conference papers. His H-index is 54, and his total citation has exceeded 19, 000, according to Google Scholar (as of August 2017). More than 10 of his publications have been each cited more than 600 times, with the highest exceeding 2, 000. His research has also been widely covered by the media, including CBS News, Scientific American, NSF News, ACM TechNews, Science Daily, The Conversation, etc. He has delivered more than 100 keynote/invited talks at conferences and universities worldwide.