CollAFL: Path Sensitive Fuzzing

Shuitao Gan[1], Chao Zhang[2]✉, Xiaojun Qin[1], Xuwen Tu[1], Kang Li[3], Zhongyu Pei[2], Zuoning Chen[4]

[1]State Key Laboratory of Mathematical Engineering and Advanced Computing. *ganshuitao@gmail.com*
[2]Institute for Network Science and Cyberspace, Tsinghua University. ✉ *chaoz@tsinghua.edu.cn*
[3]Cyber Immunity Lab. [4]National Research Center of Parallel Computer Engineering and Technology.

***Abstract*—Coverage-guided fuzzing is a widely used and effective solution to find software vulnerabilities. Tracking code coverage and utilizing it to guide fuzzing are crucial to coverage-guided fuzzers. However, tracking full and accurate path coverage is infeasible in practice due to the high instrumentation overhead. Popular fuzzers (e.g., AFL) often use *coarse* coverage information, e.g., edge hit counts stored in a compact bitmap, to achieve highly efficient greybox testing. Such inaccuracy and incompleteness in coverage introduce serious limitations to fuzzers. First, it causes *path collisions*, which prevent fuzzers from discovering potential paths that lead to new crashes. More importantly, it prevents fuzzers from making wise decisions on fuzzing strategies.**

In this paper, we propose a coverage sensitive fuzzing solution CollAFL. It mitigates path collisions by providing more accurate coverage information, while still preserving low instrumentation overhead. It also utilizes the coverage information to apply three new fuzzing strategies, promoting the speed of discovering new paths and vulnerabilities. We implemented a prototype of CollAFL based on the popular fuzzer AFL and evaluated it on 24 popular applications. The results showed that path collisions are common, i.e., up to 75% of edges could collide with others in some applications, and CollAFL could reduce the edge collision ratio to nearly zero. Moreover, armed with the three fuzzing strategies, CollAFL outperforms AFL in terms of both code coverage and vulnerability discovery. On average, CollAFL covered 20% more program paths, found 320% more unique crashes and 260% more bugs than AFL in 200 hours. In total, CollAFL found 157 new security bugs with 95 new CVEs assigned.

I. Introduction

Memory corruption vulnerabilities are the root cause of many severe threats to programs, including control flow hijacking attacks [12, 37, 38] and information leakage attacks [36]. Both defenders and attackers are eager to discover vulnerabilities in programs. Attackers rely on vulnerabilities to break target programs' execution and perform malicious actions. Defenders could patch vulnerabilities to defeat potential attacks if they could discover vulnerabilities in advance.

Coverage-guided fuzzing is one of the most popular vulnerability discovery solutions, widely deployed in industry. For example, Google's OSS-Fuzz platform [35] adopts several state-of-art coverage-guided fuzzers, including libFuzzer [34], honggfuzz [40], and AFL [45], to continuously test open source applications. It has found over 1000 bugs in 5 months [3] with thousands of virtual machines.

First of all, tracking code coverage is crucial to coverage-guided fuzzers. Accurate *path coverage* information could help fuzzers perceive all unique paths and explore them to find vulnerabilities. However, it is infeasible in practice to track path coverage, due to the extraordinarily high instrumentation overhead. Fuzzers trade off the coverage accuracy with performance. LibFuzzer [34] and honggfuzz [40] utilize the SanitizerCoverage [4] instrumentation provided by the Clang compiler, to track *block coverage*[1]. VUzzer [29] uses the dynamic binary instrumentation tool PIN [24] to track *block coverage*. AFL [45] (in GCC and LLVM mode) uses static instrumentation with a compact bitmap to track *edge coverage*, providing more information than block coverage. Even for AFL, there is a known *hash collision* issue[2], where two different edges could have a same hash and thus share a same record in the coverage bitmap. It causes a loss in the accuracy of edge coverage. Our experiments showed that up to 75% of edges could collide with others in some applications.

More importantly, utilizing coverage information to guide fuzzing is crucial to coverage-guided fuzzers. AFL utilizes the edge coverage information to identify seeds (i.e., good testcases contributing to the coverage), and adds them to a seed pool waiting for further mutation and testing. AFLfast [11] further utilizes the edge coverage information to prioritize and select seeds (from the pool) exercising less-frequent paths for mutation, to improve the efficiency of path discovery. VUzzer utilizes the block coverage information to deprioritize testcases exercising error-handling blocks and testcases exercising frequent paths. However, fuzzers fail to make the optimal decisions given the code coverage information is inaccurate. Moreover, few fuzzers utilize code coverage information to directly drive fuzzing towards non-explored paths.

To our knowledge, the consequence of coverage inaccuracy is overshadowed by the great success of fuzzers and thus has not been systematically evaluated. In this paper, we demonstrate that it actually has a crucial impact on fuzzers' abilities. We also demonstrate that, if accurate edge coverage can be achieved with low overhead and proper coverage-guided fuzzing strategies are deployed, fuzzers could significantly improve their abilities to exploring paths and finding bugs.

A. How does coverage inaccuracy blur bug finding?

First, the coverage inaccuracy could cause fuzzers fail to differentiate two different program paths in some cases. If a testcase exercising a new path that collides with a previously explored path, the fuzzer could wrongly classify it as not interesting, and miss the chance to thoroughly test this path or explore related paths. As a result, it will cause a loss in code coverage, and even miss potential vulnerabilities hidden in these paths. Similarly, the fuzzer could also wrongly classify a newly found vulnerability as not interesting, because its path collides with a previously found vulnerability.

[1]SanitizerCoverage claims supporting edge coverage in its documentation. But it is just an enhanced version of block coverage. More details will be discussed in Section II-C.

[2]http://lcamtuf.coredump.cx/afl/technical_details.txt

Second, and more importantly, the coverage inaccuracy blurs fuzzing strategies. For example, it prevents fuzzers from making optimal decisions on selecting seeds to mutate and test. For example, AFLfast [11] prioritizes seeds that exercise less-frequent paths, which may be inaccurate given the coverage is captured by an approximate bitmap. AFLgo [10] prioritizes seeds closer to target locations, requiring an accurate path coverage information too. As a result, the coverage inaccuracy issue will render these fuzzers' seed selection policies inefficient, slowing down the speed of bug finding.

B. How to improve coverage accuracy and guide fuzzers?

As aforementioned, tracking accurate path coverage is infeasible in practice, but tracking edge and block coverage are possible. The edge coverage provided by AFL provides more information than block coverage solutions. Moreover, AFL's coverage tracking solution introduces lower runtime overhead than others as well. As a result, we could port AFL's coverage tracking solution to other fuzzers, to improve their coverage accuracy. However, AFL's edge coverage itself is imperfect due to the hash collision issue. A straightforward solution to this issue is enlarging the size of the bitmap used by AFL to store coverage. As our experiments showed, this solution could not eliminate all hash collisions for known edges, but introducing a significant overhead.

This paper presents a coverage sensitive fuzzing solution *CollAFL*, which resolves AFL's hash collision issue and improves its coverage accuracy without performance loss. Furthermore, CollAFL not only leverages the accurate edge coverage information, but also uses three newly designed fuzzing strategies to drive the fuzzer towards non-explored paths, improving the efficiency of vulnerability discovery.

CollAFL resolves the *hash collision* issue in AFL by ensuring that each edge in a target program has a unique hash, so that AFL could differentiate any two edges. More specifically, we analyze the control flow graph of the target application to get a list of known edges. A carefully designed hash scheme is used to assign IDs to basic blocks and compute hashes for all edges, ensuring that the instrumentation cost is low and collisions are eliminated for known edges.

Once the hash collision issue is resolved, fuzzers can obtain accurate edge coverage information, enabling coverage sensitive fuzzing strategies, e.g., seed selection policies. Accurate edge coverage allows a fuzzer to prioritize seeds based on fine-grained properties associated to the seeds' execution paths, e.g., the number of memory-access operations, untouched neighbor-branches and untouched neighbor-descendants along each path. Correspondingly, this paper proposes three new seed selection policies: *memory-access guided*, *untouched-branch guided*, and *untouched-descendant guided*, each prioritizes the seed selection based on the path properties mentioned above. All these policies showed improvements to the fuzzer's efficiency of path and vulnerability discovery.

C. How well does coverage sensitive fuzzing perform?

We implemented a prototype of CollAFL based on the popular coverage-guided fuzzer AFL. We evaluated CollAFL on the LAVA-M dataset [14] and 24 open source applications. The evaluation results demonstrate that:

- The hash collision issue is prevalent in real world applications, up to 75% edges could be conflicted with others;
- The collision mitigation solution we proposed could resolve all hash collisions for known edges, and could help the fuzzer to explore 9.9% more code, find 250% more unique crashes and 140% more security bugs on average.
- The *untouched-branch guided* seed selection policy (together with collision mitigation) could further improve the fuzzer's efficiency of path and bug discovery. On average, it helps the fuzzer to explore 20% more code, finds 320% more unique crashes and 260% more security bugs.

In total, we have found 157 security vulnerabilities in these 24 open source applications and reported them to upstream vendors. And 23 of them have been reported by other researchers but not exposed to public. Among the remaining 134 vulnerabilities, 95 of them are confirmed by CVE. To summarize, this paper makes the following contributions:

- We studied the negative impact of coverage inaccuracy in coverage-guided fuzzers. Especially, we demonstrated that the hash collision issue in AFL severely limit its efficiency of path and vulnerability discovery.
- We designed an algorithm to resolve the hash collision issue in AFL, improving its edge coverage accuracy with a low-overhead instrumentation scheme (which is faster than AFL in most cases).
- We proposed three new coverage sensitive seed selection policies. Our empirical results confirmed that prioritizing seeds based on accurate edge coverage information significantly improves fuzzers' performance.
- We implemented a prototype CollAFL based on AFL, and evaluated it on 24 open source applications. The effectiveness of CollAFL has been partially validated by its ability to find more than a hundred new security vulnerabilities in previously well-tested applications.

II. Background and Related Work

A. Fuzzing

Fuzzing is currently the most effective and efficient state-of-art vulnerability discovery solution. In general, fuzzers will first generate a massive number of testcases to test target applications, then monitor applications' runtime executions and report bugs when security violations are detected. Fuzzers are usually easy to setup and could be scaled up to large applications. Thus fuzzing has become the dominant vulnerability discovery solution in the industry.

Fuzzers usually use two types of testcase generation solutions: grammar-based and mutation-based. Grammar-based fuzzers [15, 17] generate testcases based on known input grammar. Following the grammar, fuzzers could generate valid testcases and cover a major part of program paths. But they require much engineering work to translate input grammar, and fail to handle applications without known grammar.

On the other hand, mutation-based fuzzers [20, 34, 45] mutate existing testcases to generate new testcases without dependency on input grammar, and thus have a better scalability. Due to the simplicity and scalability, the mutation-based solution is widely adopted in practice.

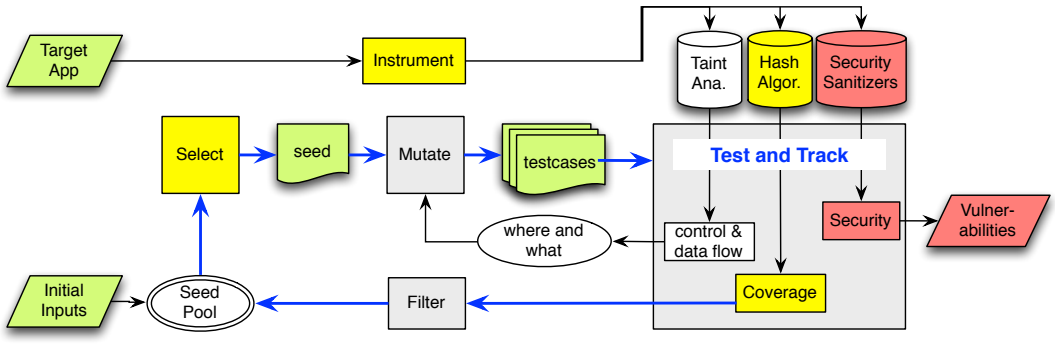

Fig. 1: The general workflow of coverage-guided fuzzing solutions.

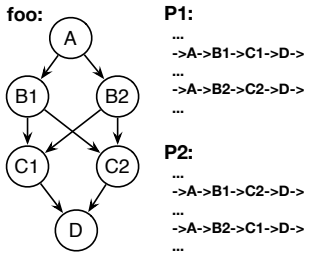

Fig. 2: Coverage inaccuracy

However, trivial mutation-based fuzzers usually have a poor code coverage. For example, they may be blocked by input format checks and could not trigger deeper vulnerabilities. Researchers thus did a lot of work on improving these fuzzers' code coverage and efficiency in fuzzing.

B. Coverage-guided Fuzzing

Towards improving the code coverage, one of the most successful solutions is coverage-guided fuzzing. It employs an evolving algorithm to drive fuzzers towards a high code coverage. AFL [45], libFuzzer [34], honggfuzz [40] and VUzzer [29] are some state-of-art coverage-guided fuzzers.

Figure 1 shows the general workflow of a coverage-guided fuzzer. It usually maintains a pool of seed testcases and performs a continuous fuzzing loop: (1) select seeds from the pool with a specific policy, (2) mutate the seeds to generate a batch of new testcases, (3) test target applications with these new testcases at a high speed, (4) monitor the program execution with instrumentations, to track its code coverage as well as security violations, (5) report vulnerabilities if security violations are detected, (6) filter good testcases contributing to code coverage and put them into the pool, and go to step 1. Following this continuous loop, fuzzers prioritize seeds contributing coverage, and evolve towards a high code coverage.

Studies have shown that *improvements to each step of this loop could promote the efficiency and effectiveness of fuzzers*. There are many factors to a fuzzer's success, including speed of testing, coverage accuracy, seed selection policies, seed mutation policies, and sensitivity to security violations etc.

For example, in step 3, several optimizations are proposed to improve the speed and throughput of fuzzers. AFL utilizes the *fork* mechanism provided by Linux to accelerate, and further adopts the *forkserver* mode and *persistent* mode to reduce the burden of fork. Moreover, AFL also supports a parallel mode[3], enabling multiple fuzzer instances to collaborate with each other. Wen Xu et.al. proposes several new primitives [44], speeding up AFL by 6.1 to 28.9 times.

In step 4, fuzzers could use different mechanisms, e.g., static instrumentation, dynamic binary instrumentation, debugging or even system emulation, to instrument target applications and track useful information. AFL utilizes GCC and Clang compilers to perform static source instrumentation, and utilizes QEMU to perform dynamic binary instrumentation. VUzzer utilizes the tool PIN [24] to perform dynamic binary instrumentation. Syzkaller [5] and kAFL [31] utilize QEMU as well as hardware features (e.g., Intel PT) to instrument.

In step 5, fuzzers often use program crashes as an indicator of vulnerabilities, because they are easy to detect even without instrumentation. However, programs do not always crash when a vulnerability is triggered, e.g., when a padding byte following an array is overwritten. Researchers have proposed several solutions to detect kinds of security violations. For example, the widely used AddressSanitizer [32] could detect buffer overflow and use-after-free vulnerabilities. There are many other sanitizers available, including UBSan [22], MemorySanitizer [39], LeakSanitizer [30], DataFlowsanitizer [2], ThreadSanitizer [33], and HexVASan [9].

In following sections, we will go over other steps in detail.

C. Coverage Tracking

Coverage-guided fuzzers utilize coverage information to drive fuzzing (step 6). As aforementioned, coverage inaccuracy blurs bug finding. So, it is essential for fuzzers to track accurate coverage. But readers should also be aware that coverage accuracy is just one factor to the success of fuzzers.

Different program paths exhibit different program behaviors and thus may have different vulnerabilities. Accurate *path coverage* could help fuzzers perceive different paths. However, it is infeasible to track all path coverage (especially the order of edges) at runtime, because the count of paths is extraordinary high and the storage overhead of each path is high.

In practice, coverage-guided fuzzers track different levels of code coverage. For example, LibFuzzer [34] and honggfuzz [40] utilize the SanitizerCoverage [4] instrumentation method provided by the Clang compiler, to track *block coverage*. VUzzer [29] uses PIN to track *block coverage*. AFL [45] uses static/dynamic instrumentation to track *edge coverage*.

Given edge coverage, we could of course infer block coverage. In some cases, we could even infer edge coverage from block coverage. SanitizerCoverage further removes *critical edges* to secure the latter inference[4], and claim to support edge coverage. But it is just an enhanced version of block coverage. Block coverage provides fewer information than edge coverage. Critical edge is just one factor that hinders inferring edge coverage from block coverage. As shown in Figure 2, there are no critical edges in function *foo*. Two

[3]https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

[4]https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage

program paths *P1* and *P2* share most of their edges, except that they take different sub-paths in function *foo*. So the block coverages of P1 and P2 are exactly the same, but their edge coverages are different. For example, the edge `B1->C1` only exists in path *P1*.

For fuzzers tracking block coverage, e.g., libFuzzer, honggfuzz and VUzzer, a solution to improve their coverage accuracy is replacing their tracking schemes with edge coverage tracking, e.g., the one used by AFL. However, the edge coverage provided by AFL is imperfect.

a) Hash Collision Issue: AFL utilizes a bitmap (default size 64KB) to track applications' edge coverage. Each byte of the bitmap represents the statistics (e.g., hit count) of a specific edge. A hash is computed for each edge, and used as the key to the bitmap. There is a hash collision issue in this scheme, i.e., two edges could have a same hash. So the fuzzer could not differentiate these edges, causing the coverage inaccuracy.

More specifically, AFL instruments the target application and assigns random keys to its basic blocks. Given an edge `A->B`, AFL then computes its hash as follows:

$$cur \oplus (prev \gg 1) \tag{1}$$

where $prev$ and cur are keys of the basic block `A` and `B` respectively. Due to the randomness of keys, two different edges could have a same hash. Moreover, the number of edges is high (i.e., comparable to the bitmap size 64K), the collision ratio would be very high, considering the birthday attack [16].

To our knowledge, the consequence of coverage inaccuracy is overshadowed by the great success of fuzzers and thus has not been systematically evaluated. Our experiments showed that as high as 75% edges could be invisible to AFL in real world applications due to the hash collision issue, greatly limiting AFL's abilities. This issue is addressed in this paper.

D. Seed Selection Policies

Recent studies [10, 11] showed that, the seed selection policy (i.e., step 1 in the fuzzing loop) is crucial to coverage-based fuzzers. A good seed selection policy could improve the fuzzer's speed of path exploring and bug findings.

AFL prioritizes seeds that are smaller and executed faster, and thus it is likely that more testcases could be tested in a given time. Honggfuzz selects seeds sequentially, and LibFuzzer prioritizes seeds that hit more new blocks. VUzzer [29] prioritizes seeds that exercise deeper paths, and deprioritizes testcases exercising error-handling blocks and frequent paths, and thus it is likely that hard-to-reach paths could be tested and useless error-handling paths will be avoided. AFLfast [11] prioritizes seeds exercising less-frequent paths and being selected fewer, and thus it is likely that cold paths could be tested thoroughly and fewer energy will be wasted on hot paths.

The seed selection policy could also strengthen the fuzzer's ability in a specific direction. For example, QTEP [42] prioritizes seeds covering more faulty code that are identified by static analysis, increasing the probability of triggering vulnerabilities during testing. SlowFuzz [27] prioritizes seeds that use more resources (e.g., CPU, memory and energy), increasing the probability of triggering algorithmic complexity vulnerabilities. AFLgo [10] prioritizes seeds that are closer to predetermined target locations (e.g., new commits waiting for reviews), enabling efficient directed fuzzing.

However, fuzzers fail to make the optimal decisions on seed selecting, given the code coverage information is not accurate. For example, AFLfast may wrongly classify a cold path as hot path if their hashes collide, and thus causes this cold path poorly tested and potential vulnerabilities missed. Moreover, few fuzzers utilize code coverage information to directly drive fuzzing towards non-explored paths. This paper proposes several new policies to address this issue.

E. Seed Mutation Policies

Mutating seeds (i.e., step 2 in the fuzzing loop) is essential to coverage-guided fuzzers. AFL and libFuzzer etc. basically use a set of deterministic and random algorithms to mutate seeds and generate new testcases. Seed mutation policies are related to several core questions: (1) sources of seeds, (2) where to mutate, and (3) what value to use for the mutation.

A set of good seeds could help generate good mutations. IMF [19] learns the order and value dependency between syscalls from normal application executions, and then generate testcases accordingly, enabling finding many deep kernel bugs. Skyfire [41] learns a probabilistic context sensitive grammar from abundant inputs to guide testcase generation. DIFUZE [13] leverages static analysis to compose *valid* inputs in the user space to test kernel drivers. Recently, researchers utilize AI techniques to help fuzzing. Patrice Godefroid et.al. proposes a RNN (Recurrent Neural Network) solution [18] to generate *valid* seed files, and could help generate inputs to pass format checks, improving the code coverage. Nicole Nichols et.al. proposes a GAN (Generative Adversarial Network) solution [26] to argument the seed pool with extra seeds, showing another promising solution. However, more research are required to further improve the quality of seed inputs.

Another core question of mutation is where to mutate. VUzzer [29] uses control-flow and data-flow features to infer bytes to mutate (e.g., magic bytes), useful for certain types of data fields. Zhiqiang et.al. proposed a solution [23] to identify sensitive bytes to mutate using static data lineage analysis. Mohit Rajpal et.al. proposes a DNN (Deep Neural Network) solution [28] to predicate which bytes to mutate, showing promising improvements. TaintScope [43] uses taint analysis to recognize checksum bytes and fix them during testing.

The other core question of mutation is what value to use for mutation. VUzzer [29] uses dynamic analysis to infer interesting values (e.g., magic numbers) to use for mutating. Honggfuzz [40] deploys a similar strategy to recognize interesting values (i.e., operands of `cmp` instructions) at runtime and greatly improve its path coverage. Laf-intel [1] transforms the target application, to split long string or constant comparison into several small comparison statements, enabling the fuzzer to find the matching mutation and exercise new paths faster.

F. Focus of this Paper

To improve fuzzers' efficiency of bug finding, we propose CollAFL, a coverage sensitive fuzzing solution. The components in yellow in Figure 1 demonstrate the focus of our solution. In short, it first improves the accuracy of code coverage tracking, and then utilizes the accurate coverage information

to guide the fuzzer, by replacing the seed selection policies. More details will be discussed in the following sections.

Research on seed mutation policies, optimizations to testing performance and instrumentation schemes, as well as fine-grained security sanitizers, are orthogonal to our proposed work. Our solution could also benefit from those work.

III. Improve Coverage Accuracy

As aforementioned, coverage inaccuracy blurs fuzzers' ability of bug finding, causing certain paths invisible to fuzzers. The first improvement of CollAFL over existing coverage-guided fuzzers is coverage accuracy. It could help fuzzers explore more paths and find more vulnerabilities.

We have studied different types of coverage granularity, and figure out edge coverage is the best choice, which has reached a good balance between instrumentation overhead and coverage accuracy. We further point out the inaccuracy issue in current edge coverage implementation, and propose a solution.

A. Coverage Granularity

There are three common types of coverage granularity, i.e., block coverage, edge coverage and path coverage. Each of them has its pros and cons.

A typical block coverage solution would track the hit count of each block during testing. It is widely adopted by fuzzers, e.g., VUzzer, libFuzzer and honggfuzz. However, it does not track the order of blocks, causing loss in coverage information. Figure 2 shows two different paths sharing the exact same number of blocks hit, and thus one of them is invisible to block-coverage fuzzers.

A typical edge coverage solution would track the hit count of each edge. A representative implementation is the one used by AFL. The instrumentation overhead is similar to block coverage solutions'. However, it does not track the order of edges, losing some information too.

Path coverage solutions will track the order of edges, providing the most complete code coverage information. However, the length of a path is very long, and the number of paths in an application is large, and thus the runtime overhead and memory overhead of tracking path coverage is extremely high. In practice, it is infeasible to track path coverage.

So, edge coverage solutions reach certain balance between efficiency and coverage information. However, even for the representative edge coverage solution AFL, there is a *hash collision issue* causing inaccuracy. CollAFL adopts the edge coverage tracking scheme and fixes the collision issue. Other fuzzers (e.g., VUzzer) could also benefit from this scheme.

B. Trivial Solution for Hash Collision

A straightforward solution to this issue is enlarging the space of hashes, i.e., the bitmap size in AFL's implementation. However, as explained by AFL itself, the current default bitmap size (i.e., 64KB) is a trade-off for performance.

The size of the map is chosen so that collisions are sporadic with almost all of the intended targets, which usually sport between 2k and 10k discoverable branch points. At the same time, its size is small enough to allow the map to be analyzed in a matter of microseconds on the receiving end, and to effortlessly fit within L2 cache.[5].

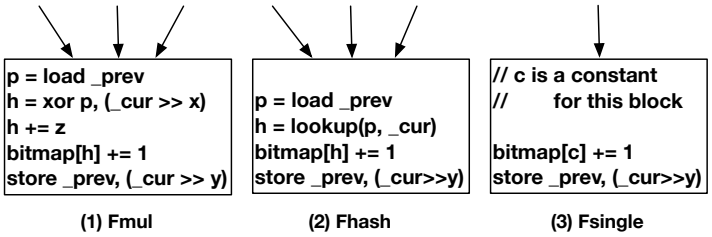

Fig. 3: Illustration of the new hash algorithms for blocks. `_cur` is the key assigned to the current block, `_prev` is a global variable for caching the key assigned to last-executed block.

We have evaluated this solution's efficiency, and confirmed that the fuzzer's performance drops quickly if we enlarge the bitmap size. As shown in Section V-A, in order to reduce the hash collision ratio to 5%, we have to increase the bitmap size from 64KB to 4MB, causing a 60% of execution speed drop-off. Even worse, we could not guarantee eliminating collisions by just enlarging the bitmap, due to the randomness. So, this is not the right solution to the hash collision issue.

C. CollAFL's Solution to Hash Collision

As shown in Equation 1, AFL uses a fixed formula to compute hash for each edge, which is fast but prone to collision. we refine it by carefully applying different hash formulas for different edges, to eliminate hash collisions while preserving the speed of hash computation and coverage tracking.

In general, given two blocks `A` and `B` with keys `prev` and `cur`, we compute the hash for edge `A->B` as follows:

$$Fmul(cur, prev) = (cur \gg x) \oplus (prev \gg y) + z \qquad (2)$$

where `<x, y, z>` are parameters to be determined, which could be different for different edges. The Equation 1 used by AFL is a specific form of this algorithm, i.e., `<x=0, y=1, z=0>` for all edges/blocks. The computation process of `Fmul` is same as AFL, having the same overhead.

As shown in Figure 3, we could choose a set of parameters for each end block, rather than each edge, to compute the edge hashes. For simplicity, a same parameter `y` will be shared between blocks, and the value $(prev \gg y)$ will be cached in the global variable $\_prev$. Each block could have a different set of parameters `<x, z>`.

Thus, given an application, we could try to find a solution of parameters for each basic block, ensuring all edges' hashes computed via `Fmul` are different. We use a greedy algorithm to search parameters of each block one by one. Once a solution to all blocks is found, we could differentiate any two edges using their hashes, and thus resolve the hash collision issue.

However, we could not guarantee to find a solution for a given application, because there are too many basic blocks in an application and we could not traverse all possible parameters. And even if we could do so, we could not guarantee a solution exists, because the keys for basic blocks are randomly assigned. Thus, we further refine the proposed hash computation algorithm as follows.

[5] http://lcamtuf.coredump.cx/afl/technical_details.txt

1) Hash Algorithm for Blocks with Single Precedent: If a block has only one precedent, as shown in Figure 3(3), we could directly assign a hash for this edge in the ending block, rather than using the Equation 2 to compute one, as long as this hash does not collide with any other edges'.

So, for a block `B` with only one precedent block `A`, we do not need to find a combination of parameters `<x, y, z>`, but just a unique hash for its sole incoming edge `A->B`. We thus introduce a different hash algorithm for it as follows:

$$Fsingle(cur, prev) : c \tag{3}$$

where `prev` and `cur` are keys assigned to block `A` and `B`, and parameter `c` is a unique constant to be determined.

This hash value `c` could be resolved offline and then hard-coded in the end block `B`. So, CollAFL is much faster than AFL to get such edges' hashes. As our experiments showed, more than 60% basic blocks have only one precedent block in most applications. So, it could save a lot of runtime overhead, improving the throughput of the fuzzer.

Furthermore, these hashes could be resolved at any time. So, to avoid conflicts, we could wait until all other edges' hashes are determined, and then pick unused hashes and assign them to blocks with only one precedent block.

2) Hash Algorithm for Blocks with Multiple Precedents: If a block `B` has multiple precedent blocks, i.e., `B` has multiple incoming edges, we have to dynamically compute the hashes in the block `B`, because the incoming edge being hit is only known at runtime. In general, we will use the aforementioned Equation 2 to compute hashes.

As discussed earlier, we could not guarantee to find a solution to this equation to avoid collision, even after removing blocks with only one precedent. We use a greedy algorithm to resolve parameters for these blocks. We denote blocks that we could resolve as *solvable blocks*, and denote blocks that we could not resolve as *unsolvable blocks*.

For an unsolvable block `B`, we introduce another hash algorithm for its incoming edge `A->B` as follows:

$$Fhash(cur, prev) : hash\_table\_lookup(cur, prev) \tag{4}$$

where `prev` and `cur` are keys of block `A` and `B`. It builds a hash table offline, with unique hashes for all edges ending with unsolvable blocks, different from all other edges' hashes. At runtime, it lookups this precomputed hash table, to get hashes for such edges, using their start and end blocks as the key.

At runtime, a hash table lookup operation is much slower than previous algorithms `Fmul` and `Fsingle`. So, we should limit the set of unsolvable blocks to as small as possible. According to our experiments, this set is usually empty.

3) Overall Mitigation Solution: First of all, we should ensure the bitmap size (i.e., size of the hash value space) is larger than the number of edges, otherwise there is no way to avoid hash collision. Then, with the three proposed hash formulas, i.e., `Fmul`, `Fsingle` and `Fhash`, we could resolve hash collisions for all edges, by applying different formulas to them depending on their types as follows:

$$F = \begin{cases} Fmul, & \texttt{Solvable blocks with multi pred} \\ Fhash, & \texttt{Unsolvable blocks with ...} \\ Fsingle, & \texttt{Blocks with single precedent} \end{cases} \tag{5}$$

Algorithm 1 The collision mitigation algorithm.

Input: Original program
Output: Instrumented program
1: (*BBS, SingleBBS, MultiBBS, Preds*) = **GetCFG**()
2: $Keys$ = **AssignUniqRandomKeysToBBs**(BBS)
3: // Fixate algorithms. Preds and Keys are common arguments
4: (*Hashes, Params, Solv, Unsolv*) = **CalcFmul**($MultiBBS$)
5: (*HashMap, FreeHashes*) = **CalcFhash**($Hashes, Unsolv$)
6: // Instrument program with coverage tracking.
7: **InstrumentFmul**(Solv, Params)
8: **InstrumentFhash**(Unsolv, HashMap)
9: **InstrumentFsingle**(SingleBBS, FreeHashes)

Algorithm 1 shows the overview of this solution.

a) **Preprocess the application***:* We first retrieve the basic block and precedent information of the target applications provided by any static analysis tool or compiler. As shown in line 1, we could get the set of basic blocks `BBS` in the program, and split it into two sub-sets `SingleBBS` and `MultiBBS`, depending on whether the block has single or multiple precedents. The precedent information of each basic block is stored in the map `Preds`. In line 2, it assigns unique random keys to each basic block in the program. This assignment information is stored in the map `Keys`.

b) **Determine algorithms for blocks***:* As shown in line 4, we first try to find proper parameters for blocks with multiple precedents using `CalcFmul`, and get the set of solvable blocks `Solv` and unsolvable blocks `Unsol`, as well as the parameters for solvable blocks `Params`, and `Hashes` taken by edges solved so far. In line 5, we build a hash map `HashMap` for unsolvable blocks `Unsol` using `CalcFhash`, and get the set of unused hashes `FreeHashes` not taken by any edge solved so far.

Algorithm 2 demonstrates the workflow of `CalcFmul`, i.e., how to search parameters for blocks with multiple precedents. It first picks a parameter `y` and then iterates each block `BB`, and traverse all combinations of `<x,z>` to find a combination, such that hashes of all edges ending with this block are different from others. If no combination could be found, this block will be classified as unsolvable and put into `Unsol`. Otherwise, the block will be put into `Solv`, and the solution will be put into `Params`. Once all basic blocks with multiple precedents are handled, and the `Unsol` set is small enough, we found a solution to the question. Otherwise, we will pick another parameter `y` and go on the previous process.

Algorithm 3 (in Appendix A) demonstrates `CalcFhash`, i.e., how to build the hash table for unsolvable blocks `Unsol`. In short, it picks random unused hashes for each edge ending with unsolvable block, and store it in the hash map `HashMap`. It also returns the set of unused hashes `FreeHashes`.

c) **Instrument blocks***:* We then instrument the application to track edge coverage, as shwon in Figure 3. For solvable blocks `Solv`, we instrument each of them with `Fmul`, i.e., same as AFL, but with different parameters `<x,y,z>` in `Params`. For unsolvable blocks `Unsolv`, we instrument each of them with `Fhash`, to search hash values in `HashMap` for edges ending with these blocks at runtime. For blocks with single precedent, we hard-code an unused hash in `FreeHashes` for each of them. In this way, we could eliminate hash collisions for all known edges.

Algorithm 2 CalcFmul

Input: *MultiBBS, Keys*[], *Preds*[]
Output: *Hashes, Params, Solv, Unsolv*

```
for y = 1 to log2MAPSIZE do
  Hashes=∅, Params=∅, Solv=∅, Unsolv=∅
  for BB in MultiBBS do
    // search parameters for BB
    for x=1 to log2MAPSIZE do
      for z=1 to log2MAPSIZE do
        tmpHashSet=∅, cur=Keys[BB]
        // hashes for all incoming edges via Equation 2
        for p ∈ Preds[BB] do
          edgeHash = (cur ≫ x) ⊕ (Keys[p] ≫ y) + z
          tmpHashSet.add(edgeHash)
        end for// iterate precedents
        // Found a solution for BB if no collision
        if sizeof(tmpHashSet) = sizeof(Preds[BB]) and
        tmpHashSet ∩ Hashes == ∅ then
          Solv.add(BB)
          Params[BB] = <x, y, z>
          Hashes.extend(tmpHashSet)
        end if
      end for// iterate z
    end for// iterate x
    Unsolv.add(BB)
  end for// iterate BB
  // Found a good solution, if Unsol is small enough.
  if sizeof(Unsol) < Δ or sizeof(Unsol)/sizeof(BBSet) < δ then
    break
  end if
end for// iterate y
return (Hashes, Params, Solv, Unsolv)
```

D. Performance analysis.

As discussed earlier, the performance overhead of these three proposed hash algorithms are as follows,

$$cost(Fhash) > cost(Fmul) > cost(Fsingle) \approx 0 \quad (6)$$

On the other hand, according to the experiments, most of the basic blocks have only one precedent block, and the number of unsolvable blocks is very small.

$$num(Fsingle) > num(Fmul) \gg num(Fhash) \approx 0 \quad (7)$$

In total, the overall performance cost introduced by the hash computation used by CollAFL is small. As shown in the evaluation Table II, our solution introduces fewer instructions and lower performance cost than AFL for most applications.

E. Implementation Details

CollAFL is built based on the edge-coverage guided fuzzer AFL. We extend AFL's *llvm_mode*, and write a Clang link time optimization pass to (1) retrieve the required basic block and edge information, (2) assign unique keys to each basic block, (3) resolve the hash computation algorithm for each basic block depending on its type, and (4) instrument each block with the hash computation and coverage tracking code. By following the algorithms in Section III-C and Figure 3, it is easy to implement the last three steps.

For the first step, we use Clang's default implementation to get the successor and precedent information, e.g., via API `llvm::TerminatorInst::getSuccessor`. However, it is an open challenge to resolve the targets of indirect control transfer offline, affecting the precision of the precedent information we need. For example, it may wrongly classify some basic blocks as single (or none) precedent blocks.

We thus take two extra steps to refine the results. First, we mark entry blocks of functions that are not directly called by anyone as multi-precedent blocks. Moreover, we unwind indirect call instructions to a set of direct calls and an indirect call instruction, similar to the de-virtualization technique [25]. It thus connects some basic blocks together, reducing the number of single-precedent blocks. As a result, we will use the `Fmul`, rather than `Fsingle`, to compute hashes for these blocks, reducing the probability of collision at runtime.

The accuracy of edge information affects the number of edges we are aware of. As our collision mitigation solution only ensures eliminating collisions for known edges, it is possible that there are some edge collisions at runtime even with CollAFL. Moreover, CollAFL currently only works for applications with source code. But it should also work on binaries, except that the edge information is less accurate. We will evaluate its performance on binaries in the future work.

IV. Prioritize Seed Selection

Existing seed selection policies mainly focus on execution speed, path frequency and path depth, but none of them focus on directly driving the fuzzer towards non-explored paths. Towards this goal, we have two intuitions that could help:

- If a path has many non-explored (or untouched) neighbor branches, then it is very likely that mutations from this path would explore those non-explored branches.
- If a path has many non-explored (or untouched) neighbor descendants, then it is very likely that mutations from this path would explore those non-explored descendants.

The eventual goal is to increase the effectiveness of vulnerability discovery. Towards this goal, we have another intuition:

- If a path has many memory access operations, it is likely to trigger potential memory corruption vulnerabilities, so do its mutations.

Mutations following these intuitions could guide the fuzzer to explore more paths and discover more vulnerabilities. We thus propose three novel seed selection policies based on these intuitions.

It is worth noting that, these policies are not limited to any fuzzer. As long as the edge coverage information is provided, we could apply these policies to the fuzzer and improve its efficiency in vulnerability discovery.

A. *Untouched-neighbour-branch guided policy*

In this policy, seeds with more *untouched neighbor branches* will be prioritized to fuzz. We believe mutations based on these seeds have a higher probability to explore those untouched neighbor branches. For simplicity, we denote this policy as **CollAFL-br**.

More specifically, we use the number of *untouched neighbor branches* as the weight of a testcase `T` as follows:

$$Weight\_Br(T) = \sum_{\substack{bb \in Path(T) \\ <bb,bb_i> \in EDGES}} IsUntouched(< bb, bb_i >) \quad (8)$$

where function $IsUntouched$ returns 1 if and only if the edge <bb, bb_i> is not covered by any previous testcase, otherwise 0.

Seeds with higher weights will be prioritized to be fuzzed in this policy. It is worth noting that, the set of previously exercised testcases will change as the testing goes on, so the return value of the function $IsUntouched$ will also change. As a result, the weight of a testcase is dynamic.

It is worth noting that, we will iterate a basic block multiple times if it is hit multiple times by the testcase. So, a block in loops will contribute more to the overall weight.

B. *Untouched-neighbour-descendant guided policy*

In this policy, seeds with more *untouched neighbor descendants* will be prioritized to fuzz. Mutations from these seeds have a higher probability to explore those untouched neighbor descendants. We denote this policy as **CollAFL-desc**.

More specifically, we will use the number of untouched neighbor descendants as the weight of a testcase `T` as follows:

$$Weight\_Desc(T) = \sum_{\substack{bb \in Path(T) \\ IsUntouched(<bb,bb_i>)}} NumDesc(bb_i) \quad (9)$$

where function $IsUntouched$ is the same as the one used in CollAFL-br policy, and function $NumDesc$ returns the number of *descendant paths* starting from the argument basic block. Its formal definition is as follows:

$$NumDesc(bb) = \sum_{<bb,bb_i> \in EDGES} NumDesc(bb_i) \quad (10)$$

The weight here is not deterministic, since the function $IsUntouched$ is dynamic. However, the number of descendant sub-paths is deterministic for each basic block. We could compute this value using static analysis, without runtime overheads. Similarly, we will iterate a basic block multiple times if it is hit multiple times by the testcase.

C. *Memory-access guided policy*

In this policy, denoted as **CollAFL-mem**, seeds with more memory access operations will be prioritized to fuzz.

More specifically, we use the number of memory access operations as the weight of a testcase `T` as follows:

$$Weight\_Mem(T) = \sum_{bb \in Path(T)} NumMemInstr(bb) \quad (11)$$

where the function $NumMemInstr$ returns the number of memory access operations in the argument basic block, which can be computed statically. As a result, the weight computed in this way is deterministic, unlike the previous two policies. Similarly, we will iterate a basic block multiple times if it is hit multiple times by the testcase.

D. *Implementation Details*

It is worth noting that, these policies could be applied to any coverage-guided fuzzer, as long as the edge coverage and block information could be provided.

We implement these three policies in AFL, by replacing its default seed selection policy. As aforementioned, we could get the number of memory access operations and the number of descendant sub-paths for each basic block at compile time.

At runtime, after a seed testcase `T` is tested, we will count its untouched neighbor branches and descendant sub-paths, as well as memory access operations along the path representatively. More specifically, we will first check the testcase's coverage bitmap, and get all edges covered by this testcase and the hit counts. Since each edge has a different hash, we could decode the start and end block of each edge from its hash. Then for each block, we will get the list of its untouched neighbor branches, based on the overall coverage bitmap. Together with the number of descendant sub-paths and memory access operations we have already collected, we could then compute the weight for all three policies accordingly.

V. Evaluation

To evaluate CollAFL, we conducted a set of experiments on different applications, to show the impact of the hash collision issue and our motivation, the code coverage and crash growth improvements, the effectiveness of bug finding of our fuzzer in real world applications, the randomness of fuzz testing, as well as the comparison between fuzzers.

We chose 24 popular open source Linux applications (in latest version when tested), including well-known tools (e.g., `nm, tcpdump, clamav`), image processing libraries (e.g., `libtiff, libexiv2`), audio and video processing (e.g., `libmpg123, libav`), and document processing (e.g., `vim, catdoc, libgxps`) etc. They are chosen based on following features: popularity in the community, development activeness, and diversity of categories. Furthermore, we also evaluated on the LAVA-M dataset [14], which has 4 applications instrumented with crafted vulnerabilities.

We evaluated four different settings of our fuzzer, including CollAFL, CollAFL-br, CollAFL-desc and CollAFL-mem, which have resolved hash collisions but applied different seed selection policies, i.e., the default policy (used by AFL), untouched-branch guided, untouched-descendant guided, and memory-access guided respectively. We further compare our fuzzer with original AFL, as well as AFL-fast [11] and CollAFL-fast (i.e., AFLfast with our collision mitigation).

We evaluated these fuzzers on these 24 applications with a same configuration, i.e., a virtual machine configured with 1 core of 2GHz Intel CPU and 1 GB RAM, running Ubuntu 15.10. We have also tested VMs with different memory sizes. It showed that 1GB is enough for the comparison and larger VM sizes do not improve these fuzzers' performance. The detail evaluation result is listed in Table VIII in Appendix B.

A. *Impact of Hash Collision*

We first evaluated the prevalence of hash collision in the state-of-art edge coverage guided fuzzer AFL, demonstrating

TABLE I: Statistics of target applications, including the file size, the number of instructions, basic blocks, and edges. In the last column, it shows the collision ratio, demonstrating the prevalence of edge hash collision.

Applications	Size	#ins.	#BB	#edges	collision
LAVA(base64)	193KB	5570	822	1308	0.8%
LAVA(uniq)	208KB	5285	890	1407	0.92%
LAVA(md5sum)	234KB	7397	1013	1560	1.02%
LAVA(who)	1.52MB	84648	1831	3332	1.8%
catdoc	202KB	6448	841	1322	1.29%
libtasn1	540KB	12511	2163	3820	2.72%
cflow	688KB	24655	4286	7001	5.2%
libncurses	338KB	21486	4646	7883	5.57%
libtiff+tiffset	1.77MB	61119	8974	14826	10.4%
libtiff+tiff2ps	1.97MB	65932	9632	15927	10.84%
libtiff+tiff2pdf	2.1MB	71530	10507	17603	12.31%
libming+listswf	4.04MB	87148	11456	19154	13.61%
libdwarf	3MB	73921	11698	20260	13.7%
tcpdump	4.62MB	127082	18781	32656	21.2%
nm	8.72MB	218326	31611	53652	36.06%
bison	3.28Mb	219268	42856	55658	32.8%
nasm	4.4MB	226665	41691	57411	33.38%
libpspp	5MB	259501	41323	71335	38.9%
objdump	11.88MB	305620	43935	74313	40.17%
clamav	11.35MB	347156	46140	81069	42.48%
exiv2+libexiv2	4.75MB	283284	59650	91287	45.87%
libsass+sassc	32.8MB	593570	68538	106738	50.7%
vim	14.7MB	478402	83877	153689	61.4%
libav	76.7MB	1776730	158009	255212	74.85%
libtorrent	97.5MB	1228513	164325	260485	75.29%

the motivation of our work. Then we evaluated the effectiveness of the trivial mitigation and our proposed solution.

1) Prevalence of Hash Collision: Table I shows the statistics of target applications tested in our experiment, whose sizes range from 100KB to 100MB. We can see that the edge hash collision ratio is very high, proportional to the number of edges. It is consistent with AFL's evaluation. For example, over 75% of edges collide with other edges in the application `libtorrent`, while it has over 260K edges.

The reason is that, the bitmap is 64KB (65.5K bytes), and thus at most 65.5K edges could be stored in it without conflicts. The remaining 194.5K (=260K-65.5K) edges must collide with other edges, no matter what hash algorithm is used. The more edges an application has, the more collisions it may have.

2) Effectiveness of the Trivial Mitigation: As aforementioned, a trivial mitigation to the hash collision issue is enlarging the bitmap. Figure 4 shows the effectiveness of this mitigation, confirming that the collision ratio drops when the bitmap size increases. For example, if we enlarge the bitmap size from 64KB to 1MB, the collision ratio of `libtorrent` drop from 75% to about 10%. However, the hash collision ratio could not be reduced to 0, even if we use a very large bitmap, due to the birthday attack and the randomness of edge hashes.

On the other hand, we also evaluated the side effects of this mitigation, pointing out that this mitigation will greatly slow down the execution speed of the fuzzer, as shown in Figure 5. It also confirms AFL's concern about the tradeoff of coverage accuracy and performance. For example, the execution speed of `libtorrent` drops off 30%, if we enlarge the bitmap size from 64KB to 1MB. The reason is that, a larger bitmap will take more time to update at runtime, and has a bad effect on the cache.

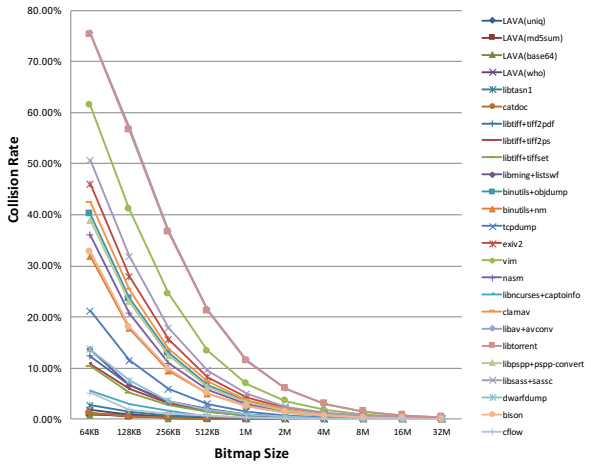

Fig. 4: Edge collision rate drops if enlarge bitmap size.

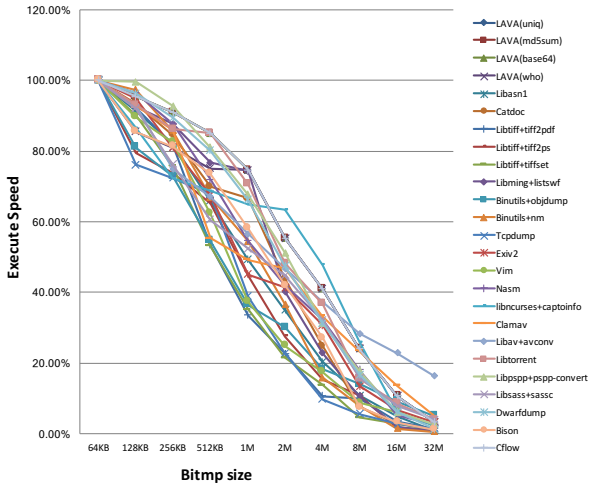

Fig. 5: Execution speed drops too if enlarge bitmap size.

The execution speed is one factor to the success of fuzzers. For example, a fork of AFL on Windows, i.e., WinAFL [8], performs much worse than AFL due to the execution speed (as well as some other restrictions). Moreover, many studies [11, 44] have shown that, the execution speed is crucial to fuzzers. But readers should also be aware that there are other factors to the success of fuzzers, in addition to speed.

So, simply enlarging the bitmap size could reduce the collision ratio (not to zero), but will slow down the execution speed, and thus slow down the speed of code coverage exploring and bug finding. As a result, this trivial mitigation is not a good solution to the hash collision issue.

3) Effectiveness of CollAFL's Mitigation: In order to eliminate collisions for known edges we use three delicate algorithms, i.e., `Fsingle, Fmul and Fhash`, to assign hashes to edges depending on the type of edges' end blocks.

Table II shows the statistics of our mitigation, including the number of instructions instrumented by AFL, the delta of instrumentations performed by CollAFL comparing to AFL, the number of instrumentations made by CollAFL in three types, and the hash collision ratio after mitigation.

TABLE II: Statistics of the collision mitigation by CollAFL.

Applications	bitmap size	AFL #ins.	CollAFL delta	Fmul	Fsingle	Fhash	coll. ratio
LAVA(uniq)	64KB	7120	-2.56%	354	536	0	0
LAVA(md5sum)	64KB	8104	-1.32%	453	560	0	0
LAVA(base64)	64KB	6576	-3.07%	310	512	0	0
LAVA(who)	64KB	14648	-8.62%	284	1547	0	0
libtasn1	64KB	17304	-4%	708	1455	0	0
catdoc	64KB	6728	-3.67%	297	544	0	0
libtiff+tiff2pdf	64KB	84048	-2.9%	4033	6473	1	0
libtiff+tiff2ps	64KB	77040	-3.18%	3590	6040	2	0
libtiff+tiffset	64KB	71792	-2.34%	3648	5326	0	0
libming+listswf	64KB	91640	-1.67%	4961	6494	1	0
objdump	128KB	351416	-2.13%	18218	25709	8	0
	64KB	-	-	16902	27033	0	11.8%
nm	64KB	252840	-1.61%	13772	17833	6	0
tcpdump	64KB	150216	-1.97%	7908	10869	4	0
exiv2	128KB	477192	-3.31%	21933	37716	1	0
	64KB	-	-	16927	42723	0	28.21%
vim	256KB	670960	-2.95%	32039	51831	7	0
	64KB	-	-	14129	69748	0	57.36%
nasm	64KB	333496	-5.83%	11130	30557	4	0
libncurses	64KB	37168	-2.93%	1779	2867	0	0
clamav	128KB	368912	-4.45%	14845	31269	26	0
	64KB	-	-	17573	28567	0	19.16%
libav	256KB	1264072	-0.6%	75068	82915	26	0
	64KB	-	-	10392	147617	0	74.32%
libtorrent	256KB	1314568	-2.91%	63012	101309	4	0
	64KB	-	-	10756	153569	0	74.84%
libpspp	128KB	330528	-3.15%	15444	25872	7	0
	64KB	-	-	16946	24377	0	8.13%
libsass	128KB	548296	-3%	26897	41640	1	0
	64KB	-	-	15785	52753	0	38.6%
libdwarf	64KB	93568	-5.03%	3494	8202	2	0
bison	64KB	342848	+1.36%	23760	19096	0	0
cflow	64KB	34288	-1.44%	1896	2390	0	0

For all applications except `bison`, more blocks are instrumented with `Fsingle` rather than `Fmul`, and the numbers of blocks instrumented with `Fhash` are close to 0. As shown in Equation 6, the cost of `Fsingle` is much lower than `Fmul`. As a result, the number of instrumentations made by CollAFL is fewer than by AFL. The fourth column proves that CollAFL instruments less instructions than AFL, for all applications except `Bison`. On average, CollAFL instruments 2.93% less instructions to applications than AFL.

It is worth noting that, CollAFL will enlarge the bitmap to a proper size (i.e., larger than the number of edges) when necessary. Results show that it successfully eliminate collisions for all known edges. Otherwise, if the bitmap size is set to 64KB, our solution could only guarantee that the bitmap is fully utilized and all 64K hashes are taken. As a result, at most 64K edges' hashes are distinct, and the remaining edges must collide with others. As shown in Table II, if we set bitmap size to 64KB, `libtorrent`'s collision ratio will be 74.84%.

So, *CollAFL could resolve collisions for all known edges, and also slightly improve the execution speed, both outperforming the trivial mitigation and the default AFL.*

B. Code Coverage

Code coverage is one of the important factors to the success of coverage-guided fuzzers. We thus evaluated different fuzzers on 24 open source applications for 200 hours, and compared their coverage performance (i.e., count of unique paths explored).

As a comparison, we also evaluated AFLfast, which demonstrated significant improvement to path discovery, and a fork CollAFL-fast that applies CollAFL's hash collision mitigation solution on AFLfast. Due to the space limit, we only show the results on 11 open source applications here. In Appendix F, we presented the results on the LAVA-M dataset.

TABLE III: Total number of paths explored in 200 hours.

Software	AFL	CollAFL	-br	-desc	-mem	AFL-fast	CollAFL-fast
Cflow	1080	+3.43%	+59.17%	+41.11%	+21.3%	1389	+7.27%
bison	1388	+9.51%	+50.94%	+75.36%	+63.04%	1969	+6.81%
tiff2pdf	5332	+5.46%	+11.7%	+14.12%	+10%	4979	+2.37%
listswf	4292	+1.34%	+6.85%	+3.36%	+0.07%	4104	+0.79%
libnurses	1529	+19.56%	+29.5%	+19.62%	+26.95%	1848	+0.6%
tiffset	1784	+0.73%	+5.04%	+10.82%	-4.37%	1616	-1.86%
exiv2	1209	+36.56%	+36.06%	+6.45%	+21.17%	201	+17.62%
libtasn1	465	+15.27%	+59.14%	+33.76%	+53.33%	511	+4.31%
libsass	8790	-1.37%	-0.61%	+3.69%	-1.66%	8771	-1.25%
nm	2389	+11.76%	-17.79%	-14.65%	-16.83%	1493	+47.15%
libpspp	2258	+6.64%	-11.43%	-4.07%	-0.27%	1772	+9.14%
Average	2774	+9.9%	+20.78%	+17.23%	+15.7%	2604	+8.45%

1) Total Coverage Improvement: As aforementioned, our hash collision mitigation could enable paths visible to fuzzers, and our seed selection policies could help explore untouched paths. So, CollAFL is very likely to improve the code coverage.

Table III shows the total number of paths in 11 applications explored by different fuzzers within 200 hours. Comparing with AFL, CollAFL (with default seed selection policy) on average found 9.9% more paths. CollAFL with the `mem`, `desc`, `br` policy on average found 15.7%, 17.23%, and 20.78% more paths respectively. Comparing with AFLfast, CollAFL-fast (with default seed selection policy) on average found 8.45% more paths.

It proves that, CollAFL (with default seed selection policy) outperforms AFL and AFLfast in terms of path discovery. Together with the three proposed seed selection policies, CollAFL could find even more paths. In other words, *the proposed hash collision mitigation solution and seed selection policies could help improve the code coverage.*

To be clear, we use the number of seeds in the seed pool to count the code coverage here. We also tried to use the coverage bitmap to count the code coverage, and showed the code coverage improvement in Table IX in the Appendix C. We could also draw a similar conclusion from this evaluation.

2) Coverage Growth over Time: Figure 7 in Appendix shows the growth of code coverage of different fuzzers on 11 open source applications within 200 hours.

From the growth trend, we can see that: (1) The code coverage of CollAFL and CollAFL-fast grows faster than AFL and AFL-fast respectively, showing that *the collision mitigation solution we applied in CollAFL is effective*. (2) The code coverage of CollAFL-br grows faster than other CollAFL settings, showing that the `br` seed selection policy is most effective. (3) All fuzzers usually could find a lot of paths at the beginning, and then get stuck at some time.

Moreover, we found that, AFLfast does not always outperform AFL in path discovery. It usually found more paths at start (e.g., first 24 hours), but then got caught up by AFL. **It is worth to evaluate the effectiveness and efficiency of fuzzers in a long time period rather than 24 hours**.

TABLE IV: Vulnerabilities detected by CollAFL* within 200 hours, including versions of target applications (latest at the time of testing), unknown and known vulnerabilities, vulnerabilities found by AFL and CollAFL* (with default, -br, -desc or -mem policy), and unknown vulnerabilities that are confirmed by CVE and could cause ACE (arbitrary code execution).

Applications	version	uniq crashes	vulnerabilities		AFL	CollAFL				unknown vulnerabilities	
			unknown	known		default	-br	-desc	-mem	CVE	ACE
libtiff	4.0.8	1569	10	3	1	7	10	8	6	7	2
libtasn1	4.12	1	1	0	0	0	0	1	0	1	0
libming	0.4.8	1303	2	4	2	2	3	4	4	2	0
libncurses	6.0	526	15	0	3	5	13	10	7	11	2
libexiv2	0.26	222	14	0	5	9	14	14	9	13	0
libsass	3.5.0	155	10	2	4	7	12	12	9	9	0
libpspp	0.10.5	412	10	2	4	5	10	10	12	6	0
bison	3.0.4	212	3	2	1	2	5	5	2	0	0
cflow	1.5	298	7	2	4	5	7	8	6	0	0
binutils	2.28	397	4	4	4	6	8	8	6	2	1
libav	12.1	239	2	0	1	1	2	2	1	2	0
tcpdump	4.9.0	10	3	0	1	2	2	3	2	2	0
clamav	0.99.2	12	1	0	0	1	1	1	1	1	0
libdwarf	20170416	14	1	0	1	1	1	1	0	1	0
libtorrent	1.1.3	177	1	0	0	1	1	1	1	1	0
nasm	2.14	1619	17	0	5	13	17	17	12	14	2
vim	8.0.679	28	3	0	1	2	3	3	2	1	1
catdoc	0.9.5	16	3	0	2	3	3	3	2	1	1
libgxps	0.2.5	32	1	0	1	1	1	1	1	1	0
Libmpg123	1.25.0	11	1	0	0	0	1	1	1	1	0
Libraw	0.18.2	14	1	0	0	0	1	1	0	1	0
Liblouis	3.2.0	38	10	0	4	5	8	7	6	7	0
Graphicmagick	1.3.26	88	4	0	2	3	4	4	3	2	0
jasper	2.0.12	122	10	4	5	7	14	14	6	9	0
Total	-	7501	134	23	51	88	141	139	99	95	9
Fraction of total vul.	-	-	85%	15%	32%	56%	90%	89%	63%	61%	4%

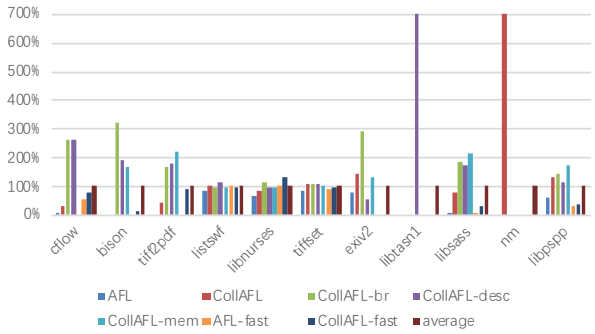

Fig. 6: Crashes found by fuzzers, comparing to average crash count.

C. Unique Crashes

In addition to code coverage, another important factor to fuzzers is the number of unique crashes found. Some crashes may be caused by a same root cause, and some are even not security vulnerabilities. But in general, the more crashes we found, there is a good probability that more vulnerabilities could be identified. Readers should be aware that fuzzers may find fewer vulnerabilities even if it could find more crashes, due to the randomness of fuzz testing.

In the same experiment shown in previous section, i.e., evaluating CollAFL, AFL and AFLfast on 24 open source applications (only 11 of them shown here) for 200 hours, we also tracked unique crashes found by different fuzzers.

1) Total Crashes Improvement: As shown in previous section, CollAFL could discover more paths than AFL. Thus it is likely that CollAFL could find more unique crashes.

Figure 6 shows the unique crashes (directly reported by AFL) in 11 applications found by different fuzzers within 200 hours. For each application, we set the average number of crashes found by all fuzzers as baseline (i.e., 100%), and present the ratio of crashes found by each fuzzer.

It shows that, CollAFL (with default seed selection policy) outperforms AFL and AFLfast in terms of finding unique crashes. Together with the three proposed seed selection policies, CollAFL could find even more crashes. In other words, the proposed hash collision mitigation solution and seed selection policies could help fuzzers find more crashes.

2) Crash Growth over Time: Figure 8 in Appendix shows the growth of crashes found by different fuzzers in 11 open source applications within 200 hours.

CollAFL outperforms AFL and AFLfast for almost all applications. For example, in the `nm` application, only CollAFL found one crash in 200 hours. CollAFL-fast outperforms AFLfast for all applications. For example, for the application `listswf`, AFLfast found more unique crashes than CollAFL-fast at first, but was surpassed at end. Again, it shows that the collision mitigation helps fuzzers find more crashes.

Moreover, CollAFL-br shows a strong growth momentum on finding crashes in all applications except `libtasn1` and `nm`. CollAFL-desc and CollAFL-mem also shows a strong growth momentum on finding crashes . In summary, all these three proposed policies could help fuzzers find more crashes, and the `br` and `desc` are the best.

Given the crashes found during fuzzing, we could further identify vulnerabilities in target applications. As discussed in Section V-D0a, we used `afl-collect` and AddressSanitizer to help de-duplicate and triage crashes. Then we manually analyzed the remaining crashes and confirmed vulnerabilities.

TABLE V: Average RSD of fuzzers among 20 trials in path discovery and crash findings.

Applications	AFL		CollAFL		-br		-desc		-mem	
	Path	Crashes	Path	Crashes	Path	Crashes	Path	Crashes	Path	Crashes
Libtasn1	1.36%	0%	1.5%	0%	2.91%	0%	0.66%	0%	1.86%	0%
Catdoc	3.1%	0%	2.47%	0%	1.97%	0%	3.71%	0%	2.88%	0%
Cflow	2.06%	8.95%	1.91%	10.5%	2.2%	12.4%	2.4%	9.6%	3.31%	12.4%
Bison	6.11%	14.82%	5.12%	20.23%	4.92%	17.92%	6.01%	15.89%	4.43%	18.92%
Tiff2pdf	2.09%	4.21%	2.22%	3.89%	2.6%	10.23%	2.6%	18.21%	4.33%	0%
Nm	7.21%	0%	5.5%	10.23%	6.1%	12.14%	5.96%	0%	5.96%	0%
libncurses	4.7%	24.7%	4.6%	46.1%	4.67%	35.8%	5.1%	43.4%	5.29%	31.9%
Libming	5.27%	12.24%	2.89%	5.2%	3.57%	8.32%	3.99%	7.47%	4.52%	3.69%
Exiv2	6.52%	20.33%	10.2%	8.76%	8.77%	9.13%	5.28%	18.25%	6.78%	13.69%
Libsass	5.02%	23.36%	5.48%	22.72%	5.13%	10.98%	3.52%	26.86%	3.91%	25.98%
Libpspp	2.34%	14.56%	1.99%	20.84%	3.03%	19.1%	3.7%	17.71%	3.1%	23.05%
Tiffset	1.49%	4.76%	2.66%	3.01%	1.57%	5.5%	3.08%	4.64%	3.3%	2.92%

TABLE VI: Deviation of fuzzers among 20 trials in bug findings. Nine specific vulnerabilities are tracked, including how many trials found them, and the shortest and longest time (in form of `hour:minute`) taken by those trials to find this vulnerability.

Fuzzer	libncurses			Libming		Exiv2		libpspp	
	Bug-5	Bug-6	Bug-7	Bug-11	Bug-12	Bug-25	Bug-26	Bug-42	Bug-43
AFL	0 (-,-)	19 (01:18, 28:29)	0 (-,-)	0 (-,-)	8 (41:19, 56:11)	0 (-,-)	0(-,-)	1 (39:45, 39:45)	0 (-,-)
AFL-pc	20 (09:32, 31:08)	18 (02:07, 08:02)	0 (-,-)	6 (34:34, 55:33)	2 (56:43, 57:51)	0 (-,-)	0(-,-)	0 (-,-)	0 (-,-)
CollAFL	20 (08:12, 19:18)	20 (00:59, 07:29)	0 (-,-)	20 (00:44, 02:56)	6 (12:23, 35:21)	0 (-,-)	0(-,-)	18 (23:01, 48:22)	0 (-,-)
-br	20 (14:18, 28:16)	20 (04:09, 12:21)	8 (45:45, 57:26)	20 (15:12, 32:15)	6 (34:02, 57:43)	8 (38:02, 56:12)	17 (21:12, 57:56)	9 (35:12, 56:21)	1 (49:44, 49:44)
-desc	20 (10:01, 18:06)	18 (06:12, 57:21)	6 (44:41, 57:33)	20 (13:12, 34:01)	7 (37:01, 57:44)	12 (28:02, 59:32)	16 (23:17, 57:05)	10 (29:18, 47:21)	0 (-,-)
-mem	20 (06:01, 11:06)	20 (00:44, 02:21)	0 (-,-)	20 (14:18, 22:21)	2 (46:02, 57:01)	0 (-,-)	6 (28:21, 46:51)	12 (28:02, 45:12)	0 (-,-)

D. Vulnerabilities

As aforementioned, we evaluated CollAFL on 24 applications. Table IV shows the detailed results of a 200 hours trial. In total, we found over 7500 unique crashes in these applications, and further identified 157 vulnerabilities.

a) Crash Triage Analysis: We used several tools to help triage crashes and filter vulnerabilities. First, we utilized the open source tool `afl-collect` [7] to de-duplicate crashes. It internally uses the GDB plugin `exploitable` [6] to compute the hash of each crash's backtrace, and then de-duplicate crashes based on the hashes. Furthermore, we recompile the target applications with AddressSanitizer, and tested them with the deduplicated crash samples. AddressSanitizer could then report the root causes of these crashes, and help us remove duplicated ones. Finally, we manually analyze the remaining crashes, and confirmed 157 unique vulnerabilities.

It is worth noting that, we only utilize AddressSanitizer to triage crashes, but do not use it in CollAFL, due to its performance overhead. So CollAFL has a good execution speed, but may miss potential vulnerabilities that do not crash. We will test CollAFL with AddressSanitizer in the future work.

We have submitted proof-of-concept samples of these 157 vulnerabilities to upstream vendors. It turned out, 23 of them are known to vendors (i.e., found by other researchers too), but have not been fixed in any release yet (i.e., unknown to us). The remaining 134 vulnerabilities are unknown to vendors, and 95 of them are acknowledged by CVE, as shown in Table X in Appendix H. Furthermore, 9 of them could cause ACE (arbitrary code execution). Non-ACE vulnerabilities could also cause serious consequences. For example, vulnerabilities in `tcpdump` and `clamav` could be utilized for DoS attacks, to disable victims' defenses.

During this trial, AFL only found 51 vulnerabilities (i.e., 32% of all detected vulnerabilities). But CollAFL with `default, mem, desc, br` seed selection policy found 88, 99, 139 and 141 (i.e., 90% of all) vulnerabilities respectively, and cover all vulnerabilities found by AFL. It shows that *CollAFL (especially with the* `br` *policy) is much better than AFL in vulnerability discovery*.

E. Randomness Evaluation

As aforementioned, we figured out some fuzzers may outperform others at start, but may lose the long-run fuzzing. A long-run evaluation could also reduce the effects of randomness, e.g., the random mutation algorithms used by fuzzers . So, in the previous sections, we run a 200 hours trial to evaluate fuzzers' code coverage, crash findings, and vulnerability discovery. But randomness is still a concern. We thus further evaluated the deviations of fuzzers' testing results.

a) Experiment Setup: However, the resource requirements is very high to do a long-run evaluation. In the 200-hour experiment we conducted, we tested 7 fuzzer settings on 24 applications. It costs about $7 * 24 * 200 = 33,600$ core hours, i.e., 1400 core days, just for one trial. So, we are not able to conduct multiple trials. For example, to evaluate the randomness, usually tens of trials are required.

Instead, we conducted 20 smaller trials. Each trial evaluated fuzzers on 12 applications for 60 hours. We evaluated in total 8 fuzzer settings on these applications. More of these settings will be discussed in the next section. It costs about $20 * 8 * 12 * 60 = 115,200$ core hours, i.e., 4800 core days.

b) Deviation in Path Discovery and Crash Findings: For each fuzzer and target application, we first sampled every 20 minute the count of path discovered so far in each trial. Then we computed the relative standard deviation (RSD)[6] among 20 trials at each sampling point. Finally, we computed the arithmetic average of RSDs of all sampling points. Table V shows the average RSD of each fuzzer among 20 trials on each application in path discovery.

[6]https://en.wikipedia.org/wiki/Coefficient_of_variation

TABLE VII: Average bugs found by fuzzers among 20 trials. Values in the parentheses are average bugs found in 60 and 24 hours respectively.

Applications	AFL	AFL-pc	AFL-laf	CollAFL	-br	-br-laf
LAVA(Base64)	(0.2, 0)	(0.1, 0)	(34.2, 32.1)	(0.8, 0.1)	(2.1, 1.8)	(39.3, 38)
LAVA (Md5sum)	(0, 0)	(0, 0)	(35.8, 31.1)	(0.4, 0)	(3.6, 0)	(48.8, 35.5)
LAVA(uniq)	(0.6, 0.2)	(1.2, 0)	(25.6, 23.3)	(1.7, 1.1)	(3.7, 0.9)	(27.7, 23.2)
LAVA(who)	(2.6, 1.2)	(2.1, 1.1)	(43.1, 35.1)	(4.1, 1.5)	(5.3, 2)	(47.7, 36.1)
Total	(3.4, 1.4)	(3.4, 1.1)	(138.7, 121.6)	(7, 2.7)	(14.7, 4.7)	(163.5, 132.8)
Cflow	(0.7, 0)	(1.1, 0.1)	(0.4, 0)	(1.8, 0.1)	(2.4, 0.2)	(2.1, 0)
bison	(0.9, 0.1)	(1, 0.5)	(0.8, 0)	(1.6, 0.5)	(1.8, 0.8)	(1.5, 0)
tiff2pdf	(0.1, 0)	(0, 0)	(0, 0)	(0.8, 0)	(1.3, 0)	(1.1, 0)
listswf	(1.8, 1)	(1.9, 0.2)	(1.1, 0)	(1.8, 1.2)	(2.1, 0.8)	(2.1, 0.4)
libnurses	(1.1, 0.9)	(1.5, 0.8)	(1.1, 0.8)	(1.6, 1.1)	(2.3, 1.9)	(2, 1.5)
tiffset	(1.9, 1.1)	(1.6, 1)	(1.8, 1.1)	(2.4, 2)	(2.8, 2)	(2.4, 1.2)
exiv2	(1.9, 0)	(2.1, 0.8)	(1.3, 0)	(2.5, 0.2)	(4.1, 1.4)	(3.1, 0.3)
libtasn1	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)
libsass	(1.7, 0)	(1.2, 0)	(1.1, 0)	(2.2, 0)	(3.4, 0)	(1.4, 0)
Nm	(0, 0)	(0, 0)	(0, 0)	(0.1, 0)	(0.3, 0)	(0, 0)
pspp	(1.2, 0.5)	(1, 0.6)	(1.1, 0)	(1.4, 0.7)	(2.3, 0.6)	(1.3, 0.1)
Total	(11.3, 3.6)	(11.4, 3.9)	(8.7, 1.9)	(16.2, 5.8)	(22.8, 7.7)	(17, 3.5)

It shows that, fuzzers' randomness in terms of path discovery and crash finding is reasonable low. In general, fuzzers' RSDs are usually smaller than 4% in path discovery, and smaller than 25% in crash findings. For example, in a bad case, for the application Exi2, AFL and CollAFL has a RSD of 6.52% and 10.2% respectively, in terms of path discovery. It is straightforward that crash finding has more randomness than path findings, because crashes could only be triggered with a probability if the vulnerable path is explored.

Moreover, the randomness in fuzzers are relatively low comparing to the improvements made by CollAFL. Table III shows that CollAFL could find 36.56% more paths than AFL for the application Exiv2. This improvement ratio is more than double of sum of AFL and CollAFL's RSDs. So, even in the worst case, CollAFL could outperform AFL in terms of path discovery. Figure 6 shows that in the worst case, CollAFL could outperform AFL in terms of crash discovery.

c) Deviation in Vulnerability Finding: As we see, fuzzers show more deviations in crash finding than path discovery. Similarly, we can infer that fuzzers have more deviations in finding vulnerabilities than crash finding.

Due to the scarcity of vulnerabilities, we did not use the sampling method in previous section to compute the RSD. Instead, we counted how many trials could find a specific vulnerability and the time used by those trials to find this vulnerability. Table VI shows the statistics results. For example, CollAFL-desc found the `Bug-6` in 18 of 20 trials. Among these 18 trials, the fastest one spends only 6 hours to find this bug, while the slowest one spends over 57 hours to find it.

It shows that, for each target vulnerability, CollAFL could find it in more trials than AFL. For example, CollAFL found the Bug-6 in all 20 trials, while AFL only found it in 19 trials. Moreover, CollAFL could find target vulnerability faster than AFL in general. For example, CollAFL used 59 seconds to find the Bug-6 in the fastest trial, while AFL used 78 seconds.

F. Comparison between Fuzzers

a) Fuzzer Settings: We tried to compare CollAFL with other AFL family fuzzers. In order to evaluate the effectiveness of our solutions, i.e., the hash mitigation as well as seed selection policies, we evaluated two extra variations of AFL.

First, we evaluated a variation of AFL denoted as `AFL-pc`, which uses the `trace-pc-guard` instrumentation method provided by Clang to track block coverage, rather than edge coverage. Second, we adopted another instrumentation solution provided by `laf-intel` [1], which divides long integer and string comparison operations into smaller ones, and splits switch statements to a group of if-else statements. We then combine `laf-intel` with AFL and CollAFL respectively, denoted as `AFL-laf` and `CollAFL-br-laf`.

b) Vulnerability Discovery: Table VII shows the average vulnerabilities found by fuzzers among 20 trials. As aforementioned, in each trial, we evaluated 8 fuzzing settings on 12 applications (and the LAVA-M dataset) for 60 hours.

First, by comparing the results in 60 hours and 24 hours, we can see that fuzzers could still find many new vulnerabilities after 24 hours.

Second, `AFL-laf` significantly outperforms AFL, AFL-pc and CollAFL in LAVA-M dataset, but underperforms them in real world applications, showing that the LAVA-M benchmark had many long string/integer comparisons hindering traditional fuzzers, but it is not the case in real world applications.

Third, `CollAFL` outperforms AFL and `AFL-pc`, but underperforms `AFL-laf` in LAVA-M, showing that this dataset focuses much on long string comparisons.

Moreover, `CollAFL-br-laf` outperforms `AFL-laf`, showing that *the seed selection policy is effective*. However, `CollAFL-br-laf` outperforms `CollAFL-br` in LAVA-M, but underperforms it in real world applications, again showing the LAVA-M dataset is specially crafted.

VI. Conclusion

In this paper, we studied the negative impact of coverage inaccuracy in coverage-guided fuzzers. We proposed a coverage sensitive fuzzing solution CollAFL, which resolves the hash collision issue in the state-of-art fuzzer AFL, enabling more accurate edge coverage information while still preserving low instrumentation overhead. Based on the accurate coverage information, we proposed three new seed selection policies to drive the fuzzer directly towards non-explored paths. Experiments showed that this solution is both effective and efficient, in terms of path discovery, crash finding and vulnerability discovery. We have found 157 new security bugs in 24 real world applications, and 95 of them are confirmed by CVE.

Acknowledgement

We would like to thank our shepherd Matthew Hicks, and the anonymous reviewers for their insightful comments. We would also thank our colleague Prof. Dong Wu at State Key Laboratory of Mathematical Engineering and Advanced Computing for his valuable comments and suggestions to the paper. This research was supported in part by the National Natural Science Foundation of China (Grant No. 61772308 61472209, and U1736209), and Young Elite Scientists Sponsorship Program by CAST (Grant No. 2016QNRC001), and award from Tsinghua Information Science And Technology National Laboratory.

REFERENCES

[1] "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[2] "Dataflowsanitizer," https://clang.llvm.org/docs/DataFlowSanitizerDesign.html.

[3] "Oss-fuzz: Five months later, and rewarding projects," https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[4] "Sanitizercoverage: Clang documentation," https://clang.llvm.org/docs/SanitizerCoverage.html.

[5] "syzkaller - kernel fuzzer," https://github.com/google/syzkaller.

[6] "The 'exploitable' GDB plugin," https://github.com/jfoote/exploitable.

[7] "Utilities for automated crash sample processing/analysis," https://github.com/rc0r/afl-utils.

[8] "WinAFL: A fork of AFL for fuzzing Windows binaries," https://github.com/ivanfratric/winafl.

[9] P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer, "Venerable variadic vulnerabilities vanquished," 2017.

[10] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *Conf. on Computer and Communication Security*, 2010.

[13] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, C. K. Shuang Hao, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Conf. on Computer and Communication Security*, 2017.

[14] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 110–121.

[15] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, p. 34, 2011.

[16] M. Girault, R. Cohen, and M. Campana, "A generalized birthday attack," in *Advances in Cryptology*. Springer, 1988, pp. 129–156.

[17] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[18] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," *CoRR*, vol. abs/1701.07232, 2017. [Online]. Available: http://arxiv.org/abs/1701.07232

[19] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Conf. on Computer and Communication Security*, 2017.

[20] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2012.

[21] M. j00ru Jurczyk, "Effective file format fuzzing," BlackHat Europe 2016.

[22] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *USENIX Security Symposium*, 2015, pp. 81–96.

[23] Z. Lin, X. Zhang, and D. Xu, "Convicting exploitable software vulnerabilities: An efficient input provenance based approach," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[25] M. Namolaru, "Devirtualization in gcc," in *Proceedings of the GCC Developers Summit*, 2006, pp. 125–133.

[26] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *CoRR*, vol. abs/1711.02807, 2017. [Online]. Available: http://arxiv.org/abs/1711.02807

[27] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Conf. on Computer and Communication Security*, 2017.

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

[29] S. Rawat, V. Jain, A. Kumar, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Network and Distributed System Security Symposium*, 2017.

[30] A. Samsonov and K. Serebryany, "New features in addresssanitizer," 2013.

[31] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *the 2012 USENIX Annual Technical Conference*, 2012, pp. 309–318.

[33] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 2009, pp. 62–71.

[34] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *Cybersecurity Development (SecDev), IEEE*. IEEE, 2016, pp. 157–157.

[35] ——, "Oss-fuzz - google's continuous fuzzing service for open source software," 2017.

[36] F. J. Serna, "The info leak era on software exploitation," in *Blackhat USA*, 2012.

[37] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Conf. on Computer and Communication Security*, 2007.

[38] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[39] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 46–55.

[40] R. Swiecki, "Honggfuzz," *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[41] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," 2017.

[42] S. Wang, J. chang Nam, and L. Tan, "Qtep: Qulity-aware test case prioritization," in *ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[43] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," 2010.

[44] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2313–2328. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134046

[45] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

APPENDIX

A. Algorithms for determining parameters for Fhash

Algorithm 3 shows the workflow of `Fhash`, i.e., how to build the hash table for unsolvable multi-precedent blocks. In short, it picks random unused hashes for each edge ending with unsolvable block, and store it in the hash map `HashMap`. It also returns the set of unused hashes `FreeHashes`.

Algorithm 3 CalcFhash

Input: *Hashes, Unsol, Keys*[], *Preds*[]
Output: *HashMap, FreeHashes*

```
1: HashMap=∅, FreeHashes= BITMAP_HASHES - Hashes
2: for BB in Unsol do
3:     cur=Keys[BB]
4:     for p ∈ Preds[BB] do
5:         HashMap(cur, Keys[p]) = FreeHashes.RandomPop()
6:     end for// iterate precedents
7: end for// iterate BB
8: return (HashMap, FreeHashes)
```

B. Effect of Memory Size on Fuzzing

We have tested fuzzers with different memory sizes. Table VIII shows the number of inputs processed by AFL per hour, in different configurations of bitmap size and memory size. It shows that, 1GB memory in general is enough for fuzz testing (unless target applications require much runtime memory).

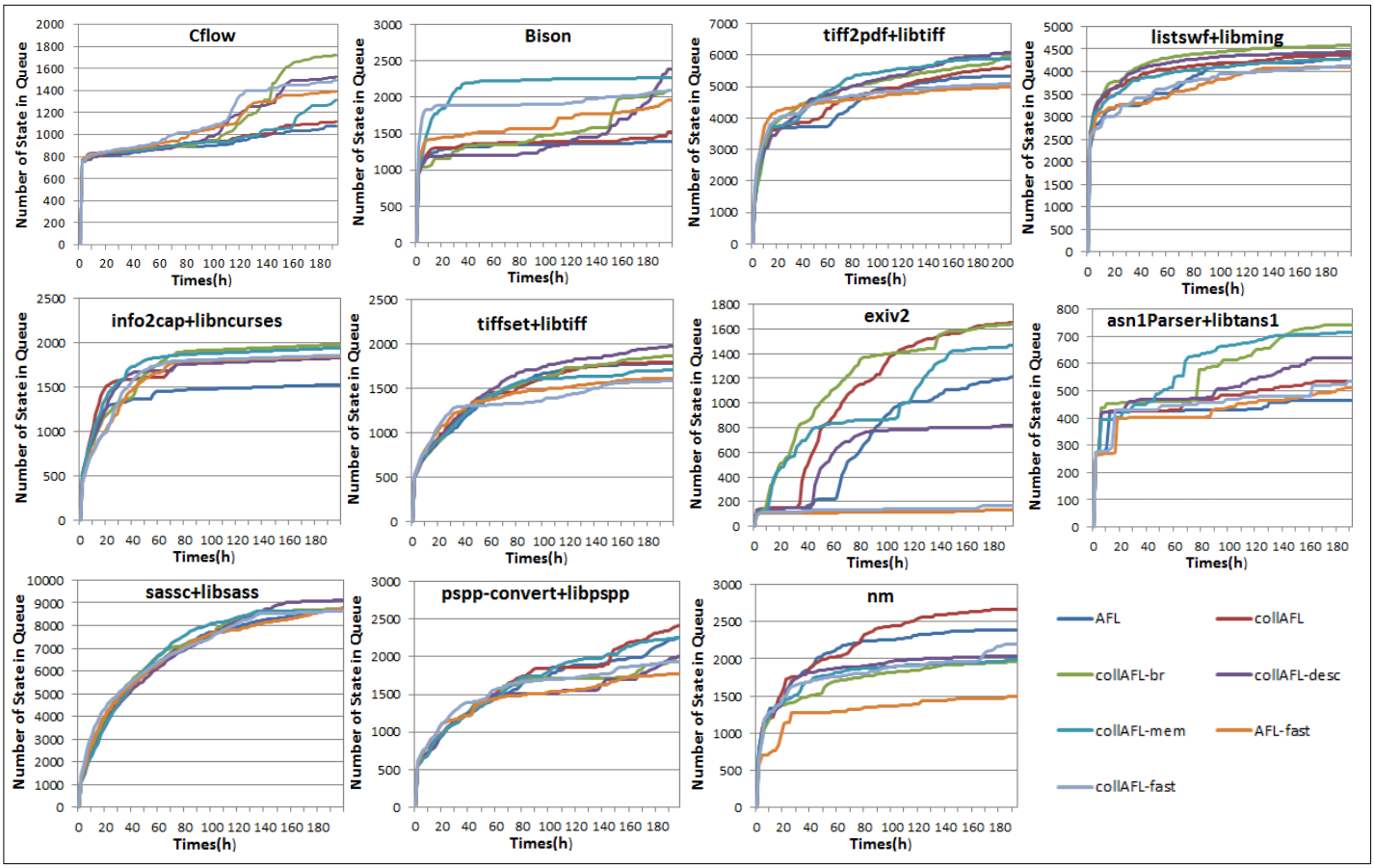

Fig. 7: Number of paths explored over time in real world applications in 200 hours.

TABLE VIII: Inputs processed by AFL per hour, given different bitmap size and memory size, running for 48 hours.

AFL Bitmap Size	64KB			256KB		
VM Memory Size	1GB	2GB	4GB	1GB	2GB	4GB
LAVA(uniq)	1,270K	1,269K	1,291K	1,099K	1,088K	1,121K
LAVA(md5sum)	778K	779K	779K	703K	712K	688K
LAVA(base64)	1,521K	1,493K	1,544K	1,103K	1,084K	1,119K
LAVA(who)	1,201K	1,205K	1,204K	1,011K	1,021K	1,034K
libtasn1	2,234K	2,146K	2,189K	1,389K	1,328K	1,361K
tiffset	1,337K	1,346K	1,365K	1,072K	1,028K	1,032K
listswf	1,329K	1,347K	1,352K	899K	933K	942K
tiff2pdf	934K	932K	923K	848K	824K	839K

C. Total Coverage Improvement in Real World Applications

As discussed in Section V-B1, we used the count of seeds in the pool as the count of paths discovered. Here, we use the coverage bitmap to count the code coverage. Table IX shows the total number of paths in 11 applications explored by different fuzzers within 200 hours. From this evaluation, we could also draw a similar conclusion as Section V-B1, i.e., CollAFL (with default seed selection policy) outperforms AFL and AFLfast in terms of path discovery. and CollAFL could do better if armed with the three proposed seed selection policies.

D. Coverage Growth in Real World Applications

As aforementioned, we evaluated CollAFL on 24 real world applications for 200 hours. Figure 7 shows the code coverage growth over time for 11 real world applications. Comparing to the LAVA-M dataset in which applications have few paths, these real world applications have much more paths. The differences between our fuzzers and the original AFL fuzzer are also more significant. This group of experiments also demonstrate similar results as the LAVA-M dataset (presented in the following sections).

TABLE IX: Total number of paths explored in real world applications in 200 hours, by recovering counters from the coverage bitmap.

Software	AFL	CollAFL	-br	-desc	-mem	AFL-fast	CollAFL-fast
Cflow	1855	+0.38%	+5.66%	+1.4%	+0.7%	1874	+0.69%
bison	2012	+8.8%	+55.82%	+45.63%	+35.34%	2903	+0.69%
tiff2pdf	7327	+1.26%	+1.08%	+5.19%	+2.14%	6770	+3.57%
listswf	3379	+0.15%	+0.53%	+0.18%	0%	3369	+0.21%
Libnurses	1986	+4.93%	+6.45%	+4.28%	+5.29%	1996	+0.8%
tiffset	3925	+0.99%	+1.83%	4.51%	-3.67%	3663	-1.56%
exiv2	7261	+1.8%	+2.16%	+1.09%	+1.45%	1599	+7.94%
libtasn1	741	+10.66%	+11.47%	+11.47%	+10.66%	747	+9.64%
libsass	23893	-0.06%	+1.54%	+1.32%	+0.44%	23125	-0.26%
Nm	3133	+5.01%	-1.02%	-2.11%	-0.7%	2667	+17.21%
libpspp	1055	+2.46%	0%	+1.23%	-3.7%	1011	+1.38%
Average	-	+3.31%	+7.77%	+6.74%	+4.36%	-	+3.66%

First, when considering the effectiveness of collision mitigation, the code coverage of CollAFL and CollAFL-fast grow faster than AFL and AFL-fast respectively, showing that

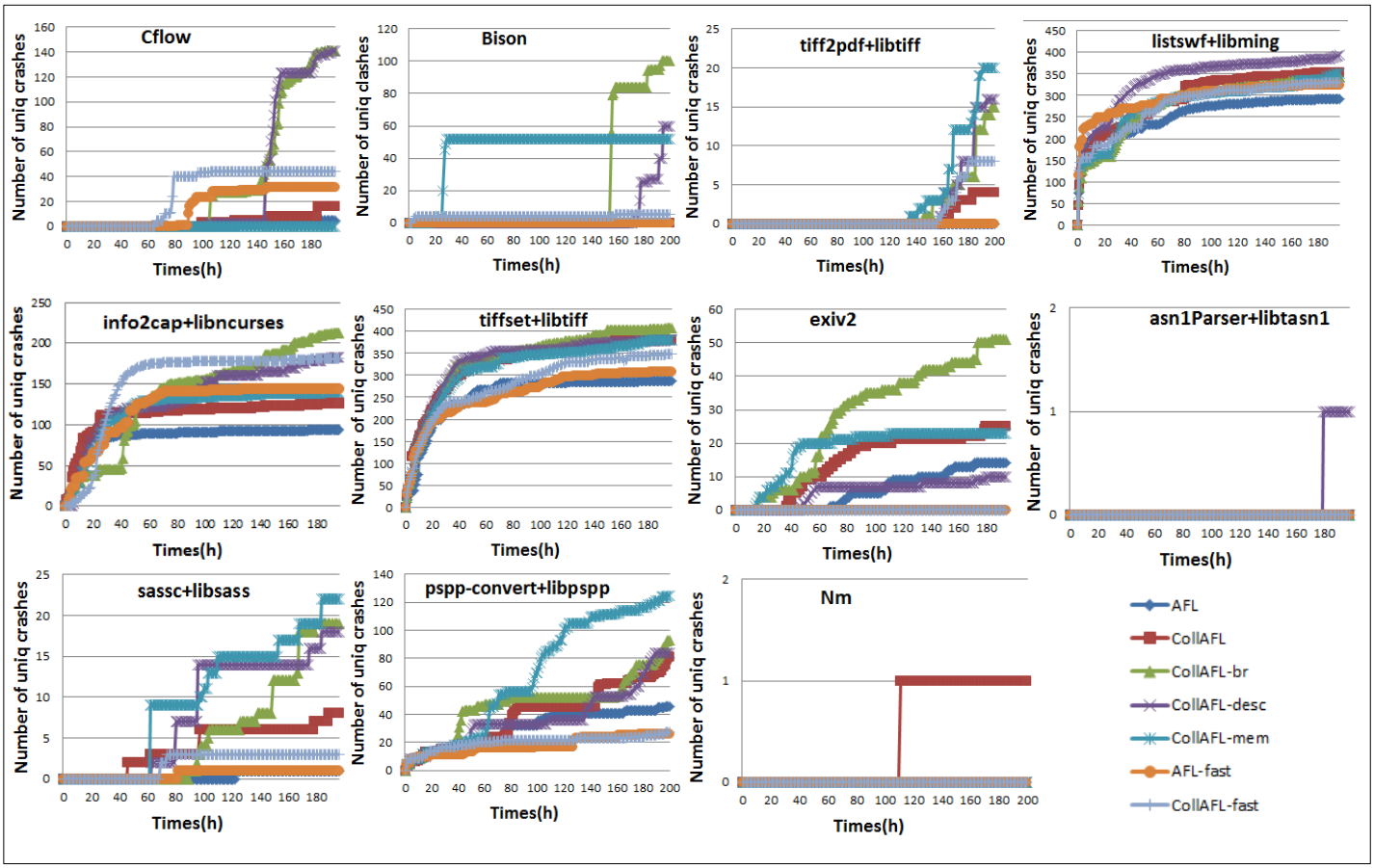

Fig. 8: Number of unique crashes found over time in real world applications in 200 hours.

the collision mitigation solution we applied in CollAFL is effective. For example, as shown in Table III, CollAFL could find 9.9% more paths than original AFL, while CollAFL-fast could find 8.45% more paths than AFLfast.

Second, the code coverage of CollAFL-br and CollAFL-desc grow faster than other settings, showing that the untouched-branch guided and untouched-descendant guided seed selection policies are better than others. For example, as shown in Table III, CollAFL-br and CollAFL-desc could find much more paths than CollAFL.

Moreover, we found that, at the beginning all fuzzers could find a lot of paths very soon, and the differences between fuzzers are not distinct. However, as time goes on, the differences between them are increasing rapidly. For most subjects, CollAFL-br and CollAFL-desc outperform the others.

It shows that a short-run trial could not fairly differentiate fuzzers, due to the randomness in fuzzing. This is also a reason that we conducted a 200-hour long experiment.

E. Crashes Growth in Real World Applications

During the 200 hours trial, we also tracked the number of crashes found by fuzzers. Figure 8 shows the number of unique crashes found over time in the real world applications in 200 hours.

CollAFL outperforms AFL for all 11 applications, and outperforms AFLfast for most applications. Especially, for the `nm` application, only CollAFL has found one crash. CollAFL-fast outperforms AFLfast for all 11 applications too. For example, in the application `listswf`, AFLfast found more unique crashes than CollAFL-fast at first, and was surpassed at end. Again, it shows that the collision mitigation solution is effective, and could help finding crashes.

Moreover, CollAFL-br shows a strong growth momentum on finding crashes in all applications except `libtasn1` and `nm`. CollAFL-desc also shows a strong growth momentum on finding crashes in all applications except `exiv2` and `libncurses`. CollAFL-mem shows a strong growth momentum on finding crashes in `libtiff`, `libpspp` and `libsass`. All these three policies could sometimes discover some unique crashes that could not be found by others in a long time. For example, CollAFL-desc found the only one unique crash in `libtasn1`.

F. Coverage Growth in LAVA-M dataset

We also evaluated CollAFL on the LAVA-M dataset for 200 hours. Figure 9 shows the code coverage growth over time. In general, the code coverage of CollAFL and CollAFL-fast grows faster than AFL and AFL-fast respectively. It proves that the collision mitigation solution in CollAFL is effective.

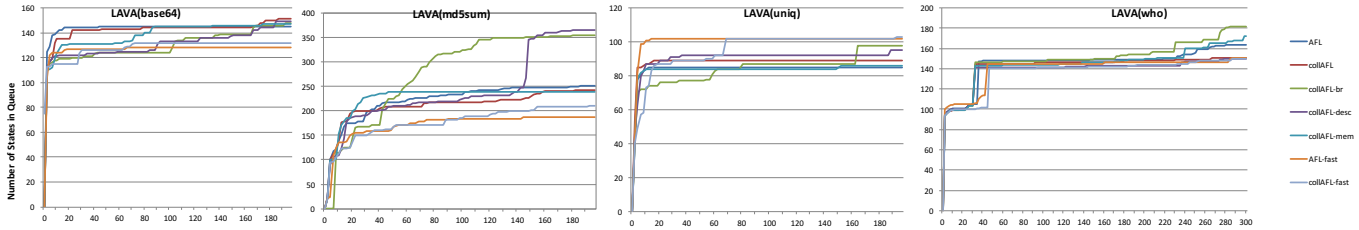

Fig. 9: Number of paths explored over time in LAVA-M dataset in 200 hours.

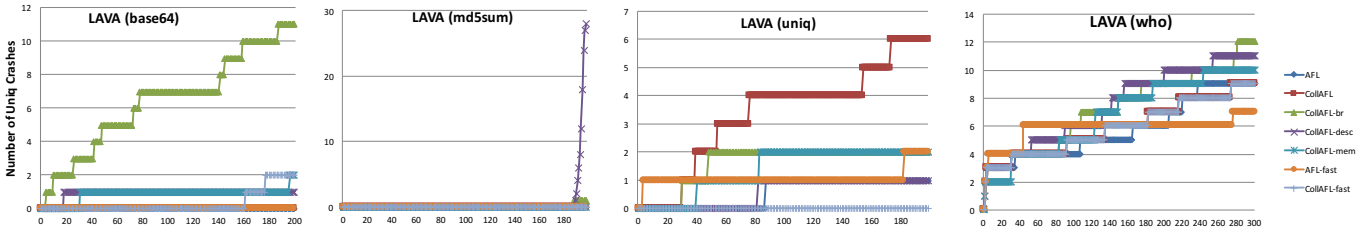

Fig. 10: Number of unique crashes found over time in the LAVA-M dataset in 200 hours.

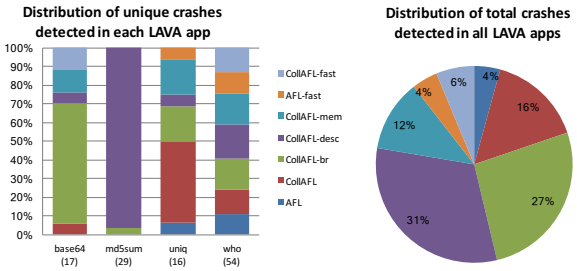

Fig. 11: Distribution of unique crashes found in the LAVA-M dataset by different fuzzers.

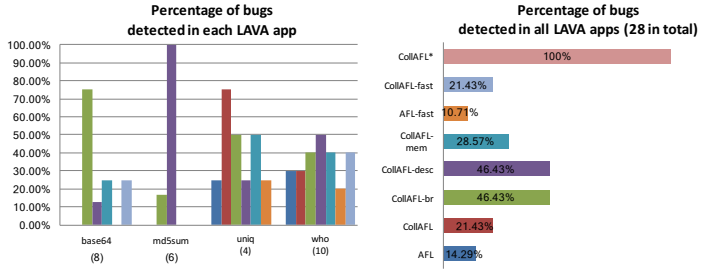

Fig. 12: Distribution of bugs found in the LAVA-M dataset by different fuzzers.

Moreover, the code coverage of CollAFL-br grows faster than others, especially for applications `md5sum` and `who`. It shows that, the `br` seed selection policy is effective, and CollAFL-br is better than other fuzzers in path discovery.

G. Crashes Found in LAVA-M dataset

During the 200 hours trial, we also tracked the number of crashes found by fuzzers. Figure 10 demonstrates the growth of crashes found. In general, CollAFL-br outperforms all others, except that it underperforms CollAFL-desc in `md5sum`.

a) Distribution of Crashes in LAVA-M: Figure 11 shows the distributions of unique crashes found in the LAVA-M dataset by different fuzzers. On the left side, it demonstrates the proportion of crashes found by each fuzzer in each application. The number below each application name in the x-axis represents the total number of unique crashes found in that application. On the right side, it demonstrates the the proportion of total crashes found by each fuzzer.

It shows that, CollAFL and CollAFL-fast could find more crashes than AFL and AFL-fast respectively. So, our hash collision mitigation improves the fuzzer's ability of crash finding, even though these applications only have about 1% of edges collisions, as shown in Table I.

It also shows that, CollAFL-br is in general better than other fuzzing solutions. So, our untouched-branch seed selection policy is better than other policies in crash finding.

b) Distribution of Vulnerabilities in LAVA-M: As aforementioned, we used the automated analysis tool `afl-collect` and AddressSanitizer to help deduplicate crashes, and finnaly performed manual analysis to confirm vulnerabilities.

Figure 12 shows the distribution of bugs found in the LAVA-M dataset by different fuzzers. On the left side, it demonstrates the distribution of bugs found by each fuzzer in each application. The number below each application name in the x-axis represents the total number of unique bugs found in that application. A same bug may be found by multiple fuzzers. On the right side, it demonstrates the overall capability of different fuzzers in bug finding.

Our fuzzer CollAFL, with the `default`, `br`, `desc` and `mem` seed selection policies, covered all vulnerabilities that are discovered during the test, including those found by AFL and AFL-fast. CollAFL-br and CollAFL-desc found over

TABLE X: The CVE vulnerabilities found by CollAFL

libtiff	CVE-2017-9935, CVE-2017-11335, CVE-2017-9936, CVE-2017-9937, CVE-2017-10688, CVE-2017-13726, CVE-2017-13727
libtasn1	CVE-2017-10790
libav	CVE-2017-9989, CVE-2017-11684
libtorrent	CVE-2017-9847
libming	CVE-2017-9987, CVE-2017-9988
objdump	CVE-2017-9955
nasm	CVE-2017-10686, CVE-2017-11111, CVE-2017-14228, CVE-2017-17812, CVE-2017-17816, CVE-2017-17817, CVE-2017-17818, CVE-2017-17813, CVE-2017-17814, CVE-2017-17810, CVE-2017-17811, CVE-2017-17820, CVE-2017-17819, CVE-2017-17815
libncurses	CVE-2017-11112, CVE-2017-10684, CVE-2017-11113, CVE-2017-10685, CVE-2017-13728, CVE-2017-13729, CVE-2017-13730, CVE-2017-13731, CVE-2017-13732, CVE-2017-13733, CVE-2017-13734
nm	CVE-2017-9954
tcpdump	CVE-2017-11108, CVE-2017-9952
exiv2	CVE-2017-9953, CVE-2017-11336, CVE-2017-11337, CVE-2017-11338, CVE-2017-11339, CVE-2017-11340, CVE-2017-11553, CVE-2017-11591, CVE-2017-11592, CVE-2017-11683, CVE-2017-12955, CVE-2017-12956, CVE-2017-12957
clamav	CVE-2017-9976
dwarfdump	CVE-2017-9998
mpg123	CVE-2017-10683
vim	CVE-2017-11109
libpspp	CVE-2017-10791, CVE-2017-10792, CVE-2017-12958, CVE-2017-12959, CVE-2017-12960, CVE-2017-12961
libsass	CVE-2017-10687, CVE-2017-11341, CVE-2017-11342, CVE-2017-11554, CVE-2017-11555, CVE-2017-11556, CVE-2017-12962, CVE-2017-12963, CVE-2017-12964
catdoc	CVE-2017-11110
libgxps	CVE-2017-11590
libraw	CVE-2017-13735
graphicsmagick	CVE-2017-13736, CVE-2017-13737
liblouis	CVE-2017-13738, CVE-2017-13739, CVE-2017-13740, CVE-2017-13741, CVE-2017-13742, CVE-2017-13743, CVE-2017-13744
jasper	CVE-2017-13745, CVE-2017-13746, CVE-2017-13747, CVE-2017-13748, CVE-2017-13749, CVE-2017-13750, CVE-2017-13751, CVE-2017-13752, CVE-2017-14229

46% bugs that are discovered during the test, while AFL and AFL-fast only found less than 15% bugs. So, our untouched-branch and untouched-descendant seed selection policies are better than others in vulnerability detection.

H. CVE Vulnerabilities

We have submitted proof-of-concept samples of these 157 vulnerabilities to upstream vendors. It turned out, 23 of them are known to vendors (i.e., found by other researchers too), but have not been fixed in any release yet (i.e., unknown to us). The remaining 134 vulnerabilities are unknown to vendors, and 95 of them are acknowledged by CVE. Table X shows the detail list of CVE vulnerabilities.

I. Discussion

1) Accuracy of Edge Information: Our solution relies on the compiler to get the edge information, and compute the hashes accordingly, in order to resolve conflicts between edges. However, it is an open challenge to get an accurate control flow graph with static analysis. Our solution only guarantee to resolve conflicts in known edges.

It is possible that the compiler may miss some edges, and CollAFL suffers some hash collisions at runtime too. But the overall collision ratio is much lower than AFL, because (1) there are no collision in the known edges, which should account for most edges; (2) the collision ratio in the unknown edges should be similar to the one in AFL.

Our solution could be extended to binary programs too. But the edge information would be much less accurate, due to the well-known challenges in bianry analysis. We can anticipate that the runtime hash collisions will increase.

However, we can make a conservative analysis, to include more (maybe untrue) edges into consideration, and thus only a small number of edges will be missed. We then could ensure eliminating collisions in this extended set of edges. As a result, we could ensure the collision ratio is very low. We will extend our solution to binaries in the future work.

2) Idel Path Coverage Sensitivity: Our solution uses the AFL's default bitmap to track edge coverage information for performance, which does not consider the order of edges. As a result, the accurage information is not ideally accurate. For example, two different paths may have the same set of edges being hit with same hit counts.

It is worth trying another solution to get more accurate path information and evaluate its effect on fuzzing. We can anticipate a slowdown in the speed of fuzzing, but it could also help the fuzzer to find paths with different calling context or order. So it could help the fuzzer to find hidden vulnerabilities as well. The biggest challenge here is to reduce the runtime overhead of tracking. We are going to explore in this direction in the future.

3) Utilizing Security Sanitizers: Our solution currently does not apply any sanitizers during fuzzing, i.e., only crashes or specific signals are reported, due to the performance consideration. It could miss potential bugs, since some bugs may not cause crashes even if they are triggered. For example, if a buffer overflow is triggered without tampering any valuable data, the program will not crash. In future work, we will apply different Sanitizers (e.g., ASAN, UBSAN, etc.) on our fuzzer and evaluate its effectiveness, in terms of both performance and efficiency of vulnerability discovery.

4) Fuzzing Experiment Setup: The seeds for fuzzing are indeed important. In order to discover more vulnerabilities, we followed the widely adopted guideline [21] to select seeds. In particular, the seeds for each application consist of two parts: benchmarks carried by the target application itself, and publicly available proof-of-concept samples that could trigger vulnerabilities in earlier versions.

When fuzzing libraries, it is also important to choose proper drivers (or front-ends). Fuzzers like libFuzzer require us to write special wrappers to test target entry functions. In our evaluation, we used the default driver programs carried with the libraries themselves. For example, the libtiff library ships with several tools, including tiff2pdf and tiff2ps. We fuzzed these tools to find potential vulnerabilities in the backend libtiff library.