Engineering a Better Fuzzer with Synergically Integrated Optimizations

Jie Liang*, Yuanliang Chen*, Mingzhe Wang*, Yu Jiang*¶, Zijiang Yang†
Chengnian Sun‡, Xun Jiao§, Jiaguang Sun*
*KLISS, BNRist, School of Software, Tsinghua University, China
†Department of Computer Science, Western Michigan University, USA
‡Cheriton School of Computer Science,University of Waterloo, Canada
§Department of Computer Science and Engineering, Villanova University, USA

***Abstract*—State-of-the-art fuzzers implement various optimizations to enhance their performance. As the optimizations reside in different stages such as input seed selection and mutation, it is tempting to combine the optimizations in different stages. However, our initial attempts demonstrate that naive combination actually worsens the performance, which explains that most optimizations are still isolated by stages and metrics.**

In this paper, we present InteFuzz, the first framework that synergically integrates multiple fuzzing optimizations. We analyze the root cause for performance degradation in naive combination, and discover optimizations conflict in coverage criteria and optimization granularity. To resolve the conflicts, we propose a novel priority-based scheduling mechanism. The dynamic integration considers both branch-based and block-based coverage feedbacks that are used by most fuzzing optimizations.

In our evaluation, we extract four optimizations from popular fuzzers such as AFLFast and FairFuzz and compare InteFuzz against naive combinations. The evaluation results show that InteFuzz outperforms the naive combination by 29% and 26% in path- and branch- coverage. Additionally, InteFuzz triggers 222 more unique crashes, and discovers 33 zero-day vulnerabilities in real-world projects with 12 registered as CVEs.

***Index Terms*—Fuzzing, Optimizations Integration**

I. INTRODUCTION

Because vulnerabilities are a major threat to software security [9], [13], [15], [16], [17], discovering them early is vital to defend against possible attacks. Widely deployed in industry, fuzzing is one of the most effective vulnerability detection techniques. For example, OSS-Fuzz [2] developed by Google continuously tests open source applications and has found over one thousand bugs in a period of five months [5]. Microsoft offers a fuzzing cloud service Springfield [1] for developers to test their software.

The main idea of fuzzing is feeding the program with invalid, unexpected or random inputs to monitor exceptions. However, Random mutation of existing inputs to construct new test cases usually produces low-quality inputs that can be easily blocked by simple format checks. Thus, coverage feedback is typically used in fuzzing [28]. There are mainly two basic types of coverage feedback, namely branch coverage used in AFL family fuzzers [6], [7], [12], [18], [29], [31] and block coverage used in libFuzzer [26], honggfuzz [4], etc.

A typical coverage feedback based fuzzing process could be divided into four stages: preparing, selecting input seeds, mutating input seeds and executing target with mutated seeds. In order to improve the performance, various optimizations [7], [12], [18], [30] have been proposed for each stage. For example, in the preparation stage, the target program is instrumented to track coverage. CollAFL [12] provides a solution to collect more accurate coverage feedback than AFL while still preserving low instrumentation overhead. In the input seed selection stage, AFLFast [7] prefers seeds that execute less visited paths, thus more effort can be used to test cold paths. In the input seed mutation stage, FairFuzz [18] automatically adjusts mutation so that the mutated inputs are more likely to execute less visited parts of the program. In the target program execution stage, some operating primitives [30] are designed to accelerate the running speed.

With so many optimizations, it is intuitive to combine and accumulate them so fuzzing can be even more practical. Indeed some optimizations have been applied in many popular fuzzers and their effect benefit all stages. For example, supplying high-quality seeds is always good for improving the fuzzing coverage. SAFL [29] optimizes the widely used fuzzer AFL by generating high-quality initial seeds and the optimization is integrated through the external input interface. Another example is vulnerability detecting optimizations that are hardened into the target binary during compilation in the preparing stage. However, these are straightforward combination because the optimizations are independent. Among the four stages, input seed selection and mutation are most essential and thus attract intensive research and engineering. There are many optimizations proposed for these two stages. Each individual optimization has been proven effective for a particular fuzzer. However, to the best of our knowledge, there has been no known work to integrate those effective optimizations in these two most important stages.

In our attempt to build a better fuzzer, we extract various optimizations and implement them in a single framework. To our surprise, simple accumulation of optimizations not only fails to improve the performance but actually leads to performance degradation. For example, we combine the input seed selection optimization of AFLFast and the input seed mutation optimization of FairFuzz and evaluate its performance on boringssl (a real program of Google fuzzer-test-suite) for 24 hours. **It turns out the fuzzer with both**

¶Yu Jiang is the correspondence author.

optimizations performs worse than either AFLFast or FairFuzz. It executes only 82% paths and 85% branches of AFLFast, and 87% paths and 92% branches of FairFuzz. Both fuzzers are AFL family tools exploiting branch-based coverage feedback to guide seed selection and mutation. Despite so many similarities, the performance degradation demonstrates that there might exist conflicts between these two optimizations. There are more diverse optimizations in other fuzzer families that exploit block-based coverage. The initial empirical study shows that it is unlikely to obtain good performance with careless optimization combination. In order to synergically integrate optimizations to gain benefits and achieve ideal results, we must examine their mechanism and address two main challenges:

1) **Develop a unified framework.** Most optimizations in the seed selection stage and seed mutation stage rely on two types of feedback – branch coverage and block coverage. These optimizations are carefully designed to work with a particular coverage criteria. A simple strategy of using a single feedback and thus replacing coverage criteria for some optimizations will not work. In order to synergically integrate these optimizations, a unified framework is needed to support both branch and block coverage criteria. Also, this framework should be compatible with optimizations in all four stages.
2) **Identify and resolve conflicts.** Our empirical study demonstrates that optimizations adopted in different fuzzing stages may conflict with each other. Such conflicts can eliminate the benefit obtained by individual optimization. Since static combination is unlikely to resolve the conflicts, we need to design dynamic algorithms to coordinate different optimizations in execution.

In this paper, we present `InteFuzz`[1], a better engineered fuzzer that integrates diverse fuzzing optimizations in multiple stages with each optimization contributing to rather than degrading the overall performance. `InteFuzz` unifies different optimizations based on two basic types of coverage feedback – branch coverage and block coverage, and records them simultaneously. It also analyzes the conflicts between optimizations in different stages. For example, a seed selection optimization based on block coverage may not be compatible with a seed mutation optimization guided by branch coverage. The seeds considered valuable by the seed selection stage may be deemed not important by the seed mutation optimization and thus ignored or badly mutated. If there exist conflicts, `InteFuzz` utilizes a priority-assigning method to resolve these conflicts. It assigns higher priority to the best optimization when conflicts occur, and the optimizations with lower priority are discarded to avoid the negative effect. The intuition is that if there are no conflicts, `InteFuzz` benefits from the joined efforts of multiple optimizations and obtains maximal performance gain. If there are conflicts, less effective optimizations are discarded so at least the improvement by the most effective single optimization is obtained.

We first evaluate `InteFuzz` on the widely used real-world program benchmark Google fuzzer-test-suite [3]. We extract typical seed selection and seed mutation optimizations in popular fuzzers and compare the results of optimization accumulation and `InteFuzz`-based optimization integration. The 24-hour experiment shows that the performance declines in all direct optimization accumulation, which demonstrates the phenomenon of optimization conflicts is normal rather than abnormal. In contrast, `InteFuzz` performs better in resolving the conflicts in all versions of integrations. Compared to optimization accumulation, the improvements of path coverage range from 13% to 29% and the branch coverage ranges from 11% to 26%. In addition, up to 222 more crashes are triggered. Furthermore, we evaluate the efficiency of `InteFuzz` on more widely used open source software from GitHub, `InteFuzz` helps find 33 more real vulnerabilities, including 12 registered as CVEs.

To summarize, this paper makes the following contributions:

- We sum up the typical types of optimizations and develop a framework `InteFuzz` to unify diverse optimizations. It supports collecting two basic types of coverage feedback and integrating individual optimizations.
- We analyze the conflicts of the optimizations working on different fuzzing stages, and design a priority-based method to resolve the conflicts when integrating.
- We implement `InteFuzz` and apply it to fuzz 8 widely used projects from GitHub. In total, we found 33 new security vulnerabilities and 12 new CVEs were assigned.

The rest of this paper is organized as follows. Section II introduces the background of fuzzing and typical optimizations in each stage. Section III gives a motivation example about conflicts. Section IV elaborates the design of `InteFuzz`. Section VI presents evaluation on the Google benchmark and some real-world projects. Section VII discusses some lessons learned in developing `InteFuzz`, and we get the conclusion in Section VIII.

II. BACKGROUND AND RELATED WORK

A. *Mutation-based fuzzing*

Mutation-based fuzzers [23] generate inputs by mutating existing test cases. This method is automatic and scalable because it is independent of the input grammar of the target program. However, aimless mutation always produces lots of worthless inputs which are easily blocked by the input checks. To improve its effectiveness, researchers utilize coverage feedback to guide fuzzing. This technique applies an evolutional strategy which filters the mutated input seeds by runtime coverage feedback. It preserves the scalability while improves the effectiveness, and it is widely used in many domains. For example, Polar [22] utilizes function code aware mutation-based fuzzing to expose many vulnerabilities of several popular ICS protocols. EVMFuzzer [11] detects many EVM vulnerabilities by mutation-based fuzzing. Beyond that, mutation-based fuzzing is also widely adopted in industry practice [1], [2], [8], [19], [20], [27].

As Figure 1 shows, mutation-based fuzzing maintains an input seed pool and mainly contains four stages: (1) Preparing the binary and initial seeds (2) Selecting input seeds from the pool (3) Mutating the selected input seeds (4) Executing the program with the mutated seeds. In the last stage, coverage

[1]https://github.com/intefuzz/intefuzz

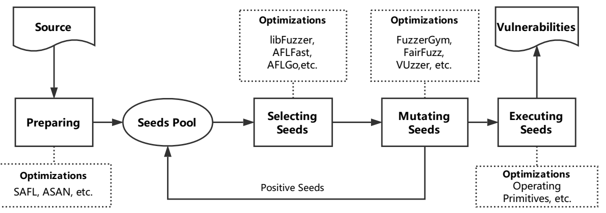


Fig. 1. Mutation-based fuzzing and corresponding optimizations, which includes four typical stages of preparing, selecting, mutating, and executing.

feedback is collected and only the seeds which have positive feedback (like covering new branches or paths) are saved to the seed pool. This coverage feedback based heuristic method helps the fuzzers explore broader parts of the target program.

There are mainly three levels of coverage feedback, namely block coverage, branch coverage and path coverage. In practice, block coverage and branch coverage are widely used. For example, libFuzzer [26] and honggfuzz [4] are all fuzzers widely used in industry and they utilize block coverage as feedback. On the other hand, AFL [31] and its family tools utilize branch coverage as feedback. These two types of coverage each have its own advantages. The branch coverage is accurate, which can find problems that exist in the transition of blocks. The block coverage is coarse, but it is easiest to track and implement among the three. The path coverage is rarely used. Because the count of paths may be extremely high, making it difficult to count and store the path coverage.

B. Fuzzing Optimizations

Optimizations in preparing and executing stage. Most optimizations in the preparing stage and the executing stage are independent of each other, and are loosely coupled with the runtime coverage feedback. They have been integrated into popular fuzzers to accumulate the improvements successfully.

To track the coverage feedback, many fuzzers perform static instrumentation [21] in the preparing binary stage. The optimization of CollAFL [12] provides a solution to collect coverage feedback without bit-map collision. To better capture errors, some vulnerability detectors are inserted into the target binary by instrumentation. For example, AddressSanitizer [25] is a widely used memory error detector which finds memory access bugs fast and effectively. In the preparing stage, initial input seeds are required. In intuition, high-quality initial seeds which execute deep places of the target program aids the fuzzing greatly [29]. Some optimizations generate these input seeds through program analysis. For example, SAFL [29] utilizes symbolic execution to supply high-quality initial seeds. To speed up the executing stage, some specific operating primitives that can improve the performance for fuzzers in a multi-core machine are designed and implemented [30].

Optimizations in selecting seeds stage. Selecting seeds is important to fuzzing and closely coupled with the run-time coverage feedback. Seed selection determines whether to mutate a seed and how many times to mutate the selected seed. A good seed selection algorithm selects suitable seeds and assigns an appropriate mutation times, which promotes expanding coverage to trigger crashes while avoids wasting resources. The simplest strategy is selecting seeds sequentially like honggfuzz [4]. On this basis, many optimizations of this stage target to expand coverage.

Some optimizations are based on block coverage feedback. For example, libFuzzer selects seeds according to piecewise constant distribution whose weight is decided by the new blocks covered by the seed. Some optimizations are based on branch coverage feedback. For example, AFL picks up the input seed which is the smallest and fastest for each covered branch. It prefers these seeds and its mutation times is calculated by some indications such as coverage and birth time. AFLFast [7] counts the path frequency based on branch coverage. It prefers to select seeds which execute less-frequent paths. Its fast strategy also gives more mutation times for these seeds, thus more energy could be used to test cold paths.

Optimizations in mutating seeds stage. Mutating seeds is the essential stage in mutation-based fuzzing, which is closely coupled with the runtime coverage feedback and interacted with the seed selection stage. In this stage, the selected seeds are mutated to generate new test inputs by a series of mutating operations like bit flipping and byte flipping. Efficient mutations would generate more valuable seeds that can pass the complicated checks and reach deep places of the code. These seeds always have some delicate structures which meet the requirements of checks. The inefficient mutation would easily damage the structure to produce low-quality seeds. The basic seed mutation knows little about the program. For example, zzuf [14] mutates the inputs by flipping random bits controlled by a predefined mutation ratio.

Some optimizations are based on block coverage feedback. For example, FuzzerGym [10] chooses the mutation operations according to the weight learned by block coverage. Some optimizations are based on branch coverage feedback. For example, FairFuzz [18] collects branch hit count to recognize rare branches. It mutates input seeds in a restricted way so that the mutated inputs are more likely to still explore the rarest branch. Accompanied with the feedback, researchers also utilize program analysis to collect more information to decide how to mutate. For example, VUzzer [24] leverages control- and data-flow features based on static and dynamic analysis to infer bytes and their values to mutate.

C. Main difference

There are a large number of optimizations in different stages of fuzzing. However, most of them focus on only improving single stage performance. Few studies systematically explore the feasibility of integrating those optimizations and how to integrate them for better performance. Generally speaking, optimizations in preparing and executing stages are independent of each other, and can be easily integrated. Meanwhile, for the stages of selecting seeds and mutating seeds, because of the close interaction and the dependence of run-time feedback, integrating optimizations in these two stages are very difficult. Moreover, a direct integration of optimizations of these two stages would result in conflicts and performance degradation. Thus, special care is required to integrate different optimizations across stages. In summary, **`InteFuzz` does not create novel individual optimizations, but it focuses on improving the overall fuzzing performance via integrating existing**

optimizations, especially those in the seed selection stage and mutation stage. It first unifies optimizations of these two stages based on two types of coverage feedback and uses an priority-assigning method to deal with the conflicts among existing optimizations.

III. Motivating example

Here, we directly integrate the seed selection optimization of AFLFast and the seed mutation optimization of FairFuzz. Then we evaluate the 24-hour performance of the direct integration on fuzzing boringssl, a Google developed communication protocol toolkit contained in Google fuzzer-test-suite.

AFLFast and FairFuzz. The seed selection of AFLFast and seed mutation of FairFuzz use two classic optimizations, namely fast power schedule selection and target branch reserved mutation. These two optimizations aim at covering the program rare parts quicker and better. The fast power schedule in AFLFast is based on path frequency defined on branch coverage feedback. It calculates mutation times for the seed according to path frequency, and prioritizes seeds exercising low-frequency paths. This optimization helps AFLFast reach the program rare branches while avoids wasting resources on hot paths. FairFuzz is based on branch hit count. When an input seed is selected, the branch which has the smallest hit count is defined as the target branch. FairFuzz mutates input seeds in a restricted way to ensure that the generated seeds still hit the target branch, and the mutated seeds whose hit count of the target branch is not smaller than the threshold will be skipped. This mutation method increases the probability to access program rare branches.

Implementation of direct integration. The direct integration is premised on the fact that the two optimizations have the same targets and work on two individual stages, and both AFLfast and FairFuzz are implemented upon AFL. We just need to copy and combine the code revisions of AFLFast and FairFuzz into the original AFL. The combined version not only collects path frequency information based on branch coverage, but also counts the hit of each branch. The seed is selected by rules of AFLFast thus the seed exercising low-frequency paths are prioritized, and the selected seed is mutated by the algorithm of FairFuzz.

Results and analysis. Figure 2 and Figure 3 show the number of paths and branches covered in 24 hours when fuzzing boringssl by AFLFast, FairFuzz and their direct integration. The results show that the direct integration performs worse than both AFLFast and FairFuzz alone. In detail, the direct integration only executes 82%, 87% paths and covers 85%, 92% branches of AFLFast and FairFuzz. The results also illustrate that the direct integration explores the program slower than AFLFast and FairFuzz.

Let us look into the example. We expect that the integration of seed selection optimization in AFLFast and seed mutation optimization in FairFuzz would improve the fuzzing performance. But the conflicts emerge and the performance degrades. The seeds selected by AFLFast might not perfectly meet the requirements of FairFuzz. In specific, FairFuzz only mutates seeds whose target branch hit count is less than the threshold, but the selected seeds from AFLFast may not cover such branches. These selected seeds are valuable to mutate

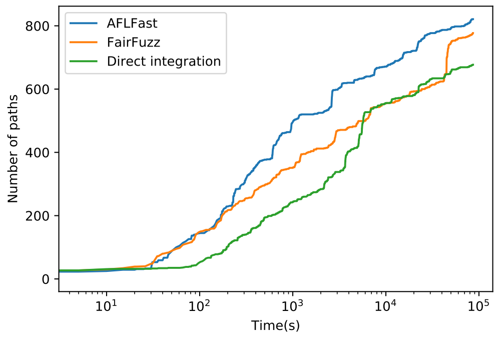


Fig. 2. Number of paths over time for fuzzing boringssl.

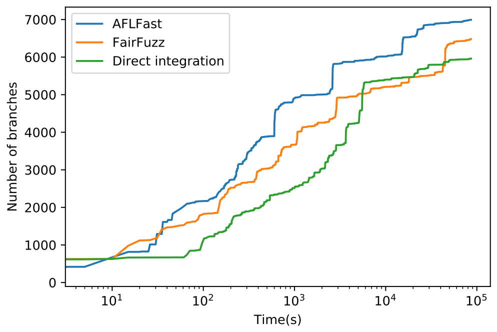


Fig. 3. Number of branches over time for fuzzing boringssl.

but are skipped in the direct integration. As a result, the speed slows down and much resources are wasted.

To solve the conflicts and take advantages of AFLFast and FairFuzz, a designated priority could help. When a seed executes the low-frequency path and covers the branch whose hit count is small enough at the same time, overall performance benefits from both. Otherwise, the priority decides whether the seed should be selected first or checked by the requirements of FairFuzz first. When seed selection has higher priority than seed mutation, the seeds are first selected based on path-frequency. Even those seeds do not meet the optimized mutation criteria of FairFuzz, they are still valuable and can be mutated by normal mutation methods of original AFL. When seed mutation has higher priority, the seeds are first checked by the criteria of FairFuzz, and the seeds that pass the check would be mutated by the optimized mutation of FairFuzz. We can set the priority to solve the conflict, in this way, at least one optimization would take effect continuously.

IV. Integration Framework Design

The proposed framework `InteFuzz` aims to integrate the optimizations in different stages, especially the closely coupled seed selection stage and seed mutation stage to obtain bigger gains. Figure 4 presents the design of `InteFuzz`. It first unifies diverse optimizations in the selection stage and mutation stage based on the basic branch and block coverage feedback. These two coverage feedback are maintained simultaneously. It then utilizes a priority-assigning method to resolve conflicts among optimizations in two stages. With support of the twofold coverage feedback and automatic conflicts resolving, `InteFuzz` integrates optimizations well and reports vulnerabilities efficiently.

A. Unify diverse optimizations

To integrate diverse optimizations, unifying them is the basic requirement. In Section II, we have introduced many kinds of optimizations. Among them, the optimizations in the loosely-coupled preparing stage and executing stage have been integrated into popular fuzzers well in practice. The difficult part is unifying and integrating the optimizations in the closely coupled seed selection stage and seed mutation stage. `InteFuzz` unifies their optimizations by simultaneously collecting twofold coverage feedback – branch coverage and block coverage, which are the basics of most optimizations in the two stages. Those minor types of optimizations with special interest such as resource consumption and time cost feedback are not considered and supported yet. As shown in the listing 1, a double record area is used to store the coverage. In specific, the first half stores the branch coverage information while the second half stores the block coverage information.

```
struct coverage_record{
    u8 branch_bit_map[MAP_SIZE];
    u8 block_bit_map[MAP_SIZE];
}
```

Listing 1. The data structure of coverage feedback record.

We use the unification and integration of selection optimization in libFuzzer and mutation optimization in FairFuzz as an example to demonstrate how it works. libFuzzer is a block-based fuzzer that selects seeds according to the weight calculated by the new block hit count of the seed. FairFuzz is a branch-based AFL family fuzzer, and it relies on restricted mutation to reserve the accessibility of program rare branches. Based on the data structure, we can unify these two optimizations as follows. libFuzzer collects block hit count from the second part of the record area to select seeds, while at the same time, FairFuzz decides the target branch for mutation based on the branch coverage from the first part of the record area. Whenever a generated seed has new coverage, `InteFuzz` updates the branch and block coverage together. In this way, the selection in libFuzzer and mutation in FairFuzz execute their own optimization logic separately and the overall performance obtains benefits from both optimizations.

The double record area structure has the following advantages: (1) It supplies the coverage feedback needed by most optimizations, which unifies diverse optimizations well. (2) It does not change any logic in optimizations, which keeps the original effectiveness of them. (3) It is convenient to implement by adding the additional logic in instrumentation of preparing stage and feedback collection of running stage.

B. Priority-assigning based conflicts resolving

When optimizations of different stages work together, they might assist each other and promote the whole fuzzing performance as described above. But sometimes these optimizations might not be harmonious and have conflicts. There are mainly two kinds of conflicts among them.

The first is the conflicts among the different criteria definitions of different optimizations. For example, libFuzzer and FairFuzz both target to expand coverage. libFuzzer selects seeds by the weight decided by the hit count of the new covered blocks. But its selection might not meet the criteria definition of FairFuzz. The mutation optimization of FairFuzz requires that the seed must hit rare branches, which means the hit count of the branch indicated block is lower than the threshold. Otherwise, it will skip them. So in some cases, libFuzzer provides suitable seeds for advanced mutation. But sometimes the seed would be skipped because of failing to meet the criteria definition of FairFuzz. In this situation, the seeds selected cannot be mutated optimally, and the mutation algorithm individually keeps waiting for its own suitable seeds.

The other is the conflicts among the different granularity of runtime information used by different optimizations. Integrating optimizations of different granularity may cause conflicts. For example, AFLFast targets to cover rare parts of the program quickly. It selects seeds according to path frequency, which is a high-level information. FairFuzz mutates seeds according to branch hit count, which is a lower level information. Because some low-frequency paths may be composed of all frequent branches, the high-level based selection of AFLFast may select some input seeds exercising low-frequency paths but cover no rare branches at all. In this situation, the advanced mutation algorithm will also skip these input seeds and wait for suitable seeds. These two types of conflicts offset the profits obtained by the integration.

ALGORITHM 1: Selection Prioritized Conflicts Resolving

```
Input : Initial seeds $S$, multi coverage map $M$
Queue ← $S$;
repeat
    foreach seed $s$ of the Pool do
        if satisfyOptimizedSelection($s$, $M$) then
            if satisfyOptimizedMutation($s$, $M$) then
                $s' = $ OptimizedMutate($s$);
            else
                $s' = $ NormalMutate($s$);
            end
        else
            continue;
        end
        runProgram($s'$);
        if causeCrash($s'$) then
            add $s'$ to $S\_c$;
        else if haveNewCoverage($s'$) then
            add $s'$ to Pool;
            updateCoverage($M$);
        end
    end
until timeout or abort-signal;
Output: Crashing seeds $S\_c$
```

To resolve conflicts and obtain bigger benefits, `InteFuzz` utilizes a priority-assigning method as following. Algorithm 1 and Algorithm 2 present two different priority-assigning solutions. The first one gives the priority to the optimized input seed selection. The optimized selection first decides whether a seed is worth to mutate. Then the selected seed is checked whether meets the requirements of the optimized mutation.

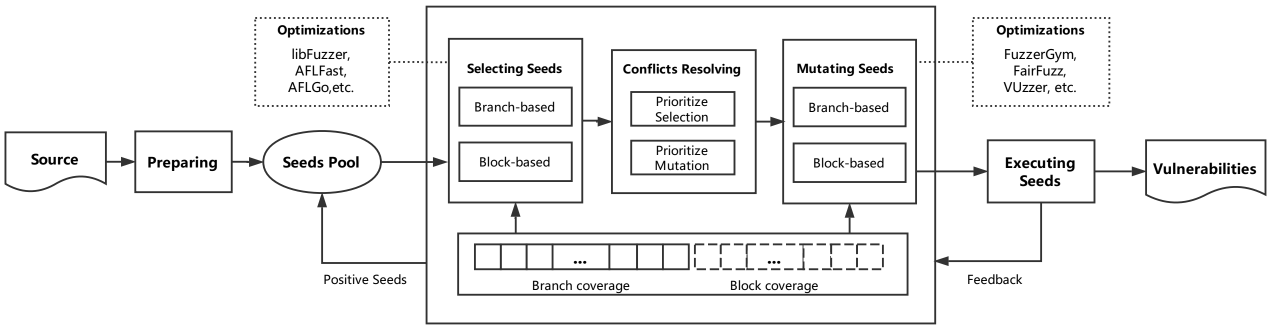


Fig. 4. InteFuzz design, which unifies diverse optimizations of selection stage and mutation stage based on recording the branch and block coverage feedback simultaneously, and integrates those optimizations together with the priority-assigning method for conflicts resolving.

If meets, it will be mutated in an optimized way. Otherwise, it would not be wasted because the normal mutation is still prepared for mutating it.

On the other hand, Algorithm 2 prioritizes the optimized mutation. Every seed will be first checked whether meets requirements by the optimized mutation. The seeds satisfying requirements will be mutated by the optimized mutation. Then the seeds left behind will be selected by the optimized selection, and the survivors will accept normal mutation.

ALGORITHM 2: Mutation Prioritized Conflicts Resolving

```
   Input  : Initial seeds $S$, multi coverage map $M$
 1 $Queue \leftarrow S$;
 2 repeat
 3     foreach seed $s$ of the $Pool$ do
 4         if satisfyOptimizedMutation($s$, $M$) then
 5             $s' =$ OptimizedMutate($s$);
 6         else if satisfyOptimizedSelection($s$, $M$)
            then
 7             $s' =$ NormalMutate($s$)
 8         end
 9         else
10             continue;
11         end
12         runProgram($s'$);
13         if causeCrash($s'$) then
14             add $s'$ to $S\_c$;
15         else if haveNewCoverage($s'$) then
16             add $s'$ to $Pool$;
17             updateCoverage($M$);
18         end
19     end
20 until timeout or abort-signal;
   Output: Crashing seeds $S\_c$
```

The priority-based solution ensures that the generated seeds which are preferred by any optimization of the two stages will be utilized even if conflicts occur. The input seeds preferred by both optimizations will be fuzzed in the best way. After assigning priority, the conflicts get resolved while the advantages of each optimized approaches are maintained.

V. IMPLEMENTATION

We extract typical optimizations of seed selection and seed mutation in popular fuzzers and implement `InteFuzz`-based optimization integrations on AFL. Original AFL uses a 64KB shared memory to record the branch coverage. Here, `InteFuzz` expands it to 128KB to record both branch and block coverage. At the compile time, `InteFuzz` carries a lightweight instrument on the source code. The code injected at every branch points is simplified as Listing 2.

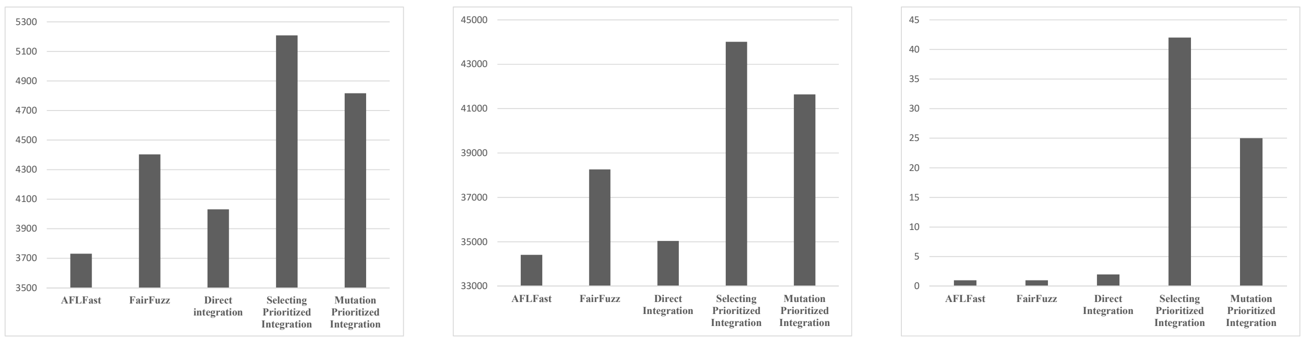
```
cur_loc=<COMPILE_TIME_RANDOM(MAP_SIZE)>;
shared_mem[cur_loc ^ prev_loc]++;
shared_mem[cur_loc + MAP_SIZE]++;
prev_loc = cur_loc >> 1;
```

Listing 2. Instrumentation code.

The value named *cur_loc* is generated randomly at compile time to identify the current basic block. Its maximum value MAP_SIZE is 64KB. A bitwise XOR is performed on *cur_loc* and *prev_loc* to identify the branch. The *shared_mem*[] array is a 128 KB shared memory region. The first half part of it records the branch coverage while the second half part maintains the block coverage. The shift operation in the last line in Listing 2 makes a distinction between the transition from block A to B and the transition from block B to A. In execution time, the process of the fuzzed program connects to the shared memory in the host process to record the coverage. With the recording of two types of feedback, `InteFuzz` supplies the basis to unify diverse optimizations.

Because most optimizations of selection stage and mutation stage locate in two types – branch-based optimization and block-based optimization, we get four versions of direct integrations – branch-based selection with branch-based mutation, branch-based selection with block-based mutation, block-based selection with branch-based mutation, and block-based selection with block-based mutation. Each of the four direct optimization integration versions corresponds to two `InteFuzz`-based conflict-aware versions.

To implement the twelve integration versions (four direct integration and eight corresponding conflict-aware integration), we first define two maps. One counts the hit number of every branch and block, and the other counts the execution time of each path. `InteFuzz` implements two input seed selection components based on branch coverage and block coverage



(a) Number of total paths (b) Number of total branches (c) Number of total crashes

Fig. 5. Performance of AFLFast, FairFuzz, Direct integration of AFLFast and FairFuzz, Selecting Prioritized Integration of AFLFast and FairFuzz and Mutation Prioritized Integration of AFLFast and FairFuzz. The conflict-aware integration outperforms the individual fuzzer and their direct integrations much.

respectively. In branch-based selection, like AFL, for each branch, it will find a smaller and faster input seed which covers it as a favored seed. For the block-based selection version, like libFuzzer, it selects a favored seed for each block. To support higher level optimizations such as path-based AFLFast, `InteFuzz` also considers the path frequency into the determination of the favored seed. In the same way, `InteFuzz` implements two input seed mutation components based on branch coverage and block coverage respectively. In the branch-based version, like FairFuzz, it determines its target branch based on branch hit count. In block-based version, it decides its target block based on block hit count. Then the target branch or block reserved mutation can be executed.

The priority-based conflicts resolving is implemented by adjusting the priority order of the seed selection and the seed mutation. As demonstrated in Algorithm 1 and 2, when the seed selection is in the front, it gets the higher priority and filters the seeds first. Otherwise, the seed mutation first checks whether the seed satisfies its requirements and directly mutates those qualified seeds.

VI. EVALUATION

We first evaluate `InteFuzz` on widely used real-world programs from Google fuzzer-test-suite. Results show that each `InteFuzz`-based conflict-aware integration outperforms their corresponding direct integration. The total improvements of path coverage range from 13% to 29%, the branch coverage range from 11% to 26%, and at most 222 more unique crashes are triggered on the benchmark. We also evaluate `InteFuzz` on more widely used open source software from GitHub, and `InteFuzz` detects 33 more real bugs, including 12 registered as CVEs.

A. Benchmark Evaluation

Experiment setup: From the widely used fuzzing benchmark Google fuzzer-test-suite, we first use the 8 real world programs that have also been used in previous literature studies for a better and fairer comparison, containing c-ares, guetzli, lcms, libssh, openssl, proj4, re2 and woff2. They are all derived from real-world libraries and contain typical bugs as well as hard-to-reach code parts. We follow the three existing metrics to evaluate the results. These metrics contain the number of executed paths, covered branches and triggered unique crashes. The first two metrics evaluate the coverage of the target programs, and the last metric reveals the probability of detecting vulnerabilities. The crashes are distinguished by execution paths. Thus several unique crashes might point to the same bug. The more unique crashes we detected, the higher probability and more vulnerabilities could be identified.

To quantify whether `InteFuzz`-based conflict-aware integration improves the corresponding direct integration or not, we carry out 4 groups of integration experiments (**br**anch-based selection with **br**anch-based mutation–**BrBr**, **br**anch-based selection with **bl**ock-based mutation–**BrBl**, **bl**ock-based selection with **br**anch-based mutation–**BlBr**, and **bl**ock-based selection with **bl**ock-based mutation–**BlBl**). In each group (e.g. **BrBr**), the direct integration version (e.g. **Direct-BrBr**) is compared to the two conflict-aware integrations with selection prioritized version (e.g. **InteFuzz-BrBrS**) and mutation prioritized version (e.g. **InteFuzz-BrBrM**).

We conduct 24-hour experiments of each tool in single core mode on a 64-bit machine with 32 cores (Intel(R) Xeon(R) CPU E5- 2630 v3 @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04 as host OS. Each fuzzer is supplied with the same set of initial input seeds and instrumented with AddressSanitizer. Table I, Table II, Table III and Figure 5 present the number of paths executed, branches covered and crashes triggered by AFLFast, FairFuzz, direct integrations and corresponding `InteFuzz`-based conflict-aware integration versions.

*1) **BrBr Group** – branch-based selection and branch-based mutation:* This group represents the type which integrates optimizations in branch-based selection and branch-based mutation. For example, integrate the selection optimizations of AFLFast and mutation optimizations of FairFuzz.

Figure 5 compares the overall performance of AFLFast, FairFuzz, their direct integration version **Direct-BrBr** and `InteFuzz`-based conflict-aware integration with selection prioritized version **InteFuzz-BrBrS** and mutation prioritized version **InteFuzz-BrBrM**. Figure 5 (a), Figure 5 (b) and Figure 5 (c) show that the coverage performance of direct

TABLE I
NUMBER OF PATHS

Project	AFLFast	FairFuzz	Direct-BrBr	InteFuzz-BrBrS	InteFuzz-BrBrM	Direct-BrBl	InteFuzz-BrBlS	InteFuzz-BrBlM	Direct-BlBr	InteFuzz-BlBrS	InteFuzz-BlBrM	Direct-BlBl	InteFuzz-BlBlS	InteFuzz-BlBlM
c-ares	26	26	22	37	38	21	37	36	23	35	38	24	41	37
guetzlili	4	4	4	699	307	4	349	308	4	391	326	275	328	334
Lcms-	108	272	103	326	342	77	346	368	260	370	323	106	394	365
libssh	20	222	20	22	20	19	20	22	20	22	21	19	22	22
openssl	881	1068	926	1113	1070	1116	1095	1083	1045	1046	1102	919	1060	1078
proj4	45	49	52	76	80	44	81	74	43	79	71	51	70	72
re2	2645	2760	2903	2932	2957	2646	2903	2898	2618	2877	2914	2784	2800	2959
woff2	2	2	2	3	3	2	3	3	2	3	3	2	3	3
Total	3731	4403	4032	5208	4817	3929	4834	4792	4015	4823	4798	4180	4718	4870
Improvement	-	-	-8% ↓	29% ↑	19% ↑	-11% ↓	23% ↑	22% ↑	-9% ↓	20% ↑	20% ↑	-5% ↓	13% ↑	17% ↑

TABLE II
NUMBER OF BRANCHES

Project	AFLFast	FairFuzz	Direct-BrBr[1]	InteFuzz-BrBrS	InteFuzz-BrBrM	Direct-BrBl[2]	InteFuzz-BrBlS	InteFuzz-BrBlM	Direct-BlBr[3]	InteFuzz-BlBrS	InteFuzz-BlBrM	Direct-BlBl[4]	InteFuzz-BlBlS	InteFuzz-BlBlM
c-ares	149	149	145	198	198	141	198	198	144	198	198	156	198	198
guetzlili	53	53	53	4429	2355	53	2373	2265	53	2833	2341	2393	2337	2364
Lcms-	1260	4578	1243	5175	5039	1149	5071	5267	4137	5245	5077	1250	6079	5240
libssh	1065	1065	1065	1067	1067	1065	1067	1067	1063	1067	1067	1065	1067	1067
openssl	8827	9206	8997	9285	9298	9257	9273	9298	9139	9230	9290	8863	9222	9251
proj4	303	254	258	377	379	223	378	379	263	379	377	306	377	377
re2	22722	22927	23249	23444	23272	22596	23241	23205	22601	23171	23308	22948	22834	23519
woff2	30	30	30	32	32	30	32	32	30	32	32	30	32	32
Total	34409	38262	35040	44007	41640	34514	41633	41711	37430	42155	41690	37011	42146	42048
Improvement	-	-	-8% ↓	26% ↑	19% ↑	-10% ↓	21% ↑	21% ↑	-2% ↓	13% ↑	11% ↑	-3% ↓	14% ↑	14% ↑

TABLE III
NUMBER OF CRASHES

Project	AFLFast	FairFuzz	Direct-BrBr	InteFuzz-BrBrS	InteFuzz-BrBrM	Direct-BrBl	InteFuzz-BrBlS	InteFuzz-BrBlM	Direct-BlBr	InteFuzz-BlBrS	InteFuzz-BlBrM	Direct-BlBl	InteFuzz-BlBlS	InteFuzz-BlBlM
c-ares	0	0	0	8	9	0	9	8	0	4	7	0	5	10
guetzlili	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Lcms-	0	0	0	0	0	0	0	0	0	0	0	0	0	0
libssh	0	0	0	0	0	0	0	0	0	0	0	0	0	0
openssl	1	1	2	34	16	40	51	22	5	11	11	2	8	9
proj4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
re2	0	0	0	0	0	0	163	0	0	0	0	0	0	0
woff2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	1	1	2	42	25	40	223	30	5	15	18	2	13	19
Improvement	-	-	1 ↑	41 ↑	24 ↑	39 ↑	222 ↑	29 ↑	4 ↑	14 ↑	17 ↑	1 ↑	12 ↑	18 ↑

integration does not get obvious improvement compared to AFLFast and FairFuzz, but the performance degradation of the direct integration especially compared to FairFuzz alone is amazing. The conflicts between AFLFast and FairFuzz are both caused by the different criteria definitions and granularity of runtime information. AFLFast optimizes fuzzing by only selecting input seeds which execute low-frequency path while FairFuzz only mutates seeds which cover rare branches. The path frequency is based on branch feedback, however, low-frequency paths may be composed of all frequent branches. Thus conflicts emerge and FairFuzz has to keep waiting for suitable seeds. The two `InteFuzz`-based conflict-aware integrations perform much better than AFLFast, FairFuzz and their direct integration in executed paths and covered branches, and are more powerful to trigger unique crashes.

We use the statistical data of each project from the first four columns of Table I, Table II, and Table III for more detailed comparisons. The direct integration **Direct-BrBr** executes 8% more paths and covers 2% more branches than AFLFast. It only execute 92% paths and covers 92% branches of FairFuzz. But luckily it triggers 2 unique crashes, and AFLFast and FairFuzz only trigger 1 unique crash. From the fifth column of these three tables, we find that selection prioritized version **InteFuzz-BrBrS** executes 40%, 18%, and 29% more paths and covers 28%, 15%, and 26% more branches than AFLFast, FairFuzz and their direct integration, respectively. From the sixth column, we observe that mutation prioritized version **InteFuzz-BrBrM** executes 29%, 9%, and 19% more paths and covers 21%, 9%, and 19% more branches than AFLFast, FairFuzz and their direct integration, respectively. Furthermore, **InteFuzz-BrBrS** triggers 42 crashes and **InteFuzz-BrBrM** triggers 25 crashes. The results demonstrate that the `InteFuzz`-based conflict-aware integration of branch-based selection optimization and branch-based mutation optimization improves the fuzzing performance.

2) ***BrBl Group*** *– branch-based selection and block-based mutation:* This group represents the mixed type which integrates optimizations in branch-based selection and block-based mutation.

From the seventh to ninth columns of the tables, we find that the performance of the direct integration **Direct-BrBl** degrades, and the selection prioritized version **InteFuzz-BrBlS** and the mutation prioritized version **InteFuzz-BrBlM** outperform others. In detail, the selection prioritized version and the mutation prioritized version execute 23% and 22% more paths and cover 21% and 21% more branches than the direct integration, respectively. The performance degradation shows the existence of conflicts. In this situation, the conflicts are caused by different criteria definitions and different granularity of runtime information. The selection optimized fuzzer selects seeds according to path frequency based on branch coverage while the other fuzzer chooses mutation operation based on block coverage. Thus the selected seeds may not meet the criteria definitions of the other. Besides, different granularities cause the integrated fuzzer cannot always find the most suitable mutation operations. For the number of crashes triggered, the selection prioritized version triggers 223 crashes in two projects – openssl and c-areas, while the direct integration triggers 40 crashes only in only one project openssl. The mutation prioritized version triggers 30 crashes. This number is smaller than the direct integration, but it also detects crashes in two projects openssl and c-ares. The results demonstrate that the `InteFuzz`-based conflict-aware integration of branch-based selection optimization and block-based mutation optimization improves the fuzzing performance.

3) ***BlBr Group*** *– block-based selection and branch-based mutation:* This group represents the mixed type which integrates optimizations in block-based selection and branch-based mutation. For example, integrate selection optimizations of libFuzzer and mutation optimizations of FairFuzz.

From the tenth to twelfth columns of the tables, we observe that the performance of the direct integration **Direct-BlBr** degrades, and the selection prioritized version **InteFuzz-BlBrS** and the mutation prioritized version **InteFuzz-BlBrM** outperform others. In detail, the selection prioritized version and the mutation prioritized version execute 20% and 20% more paths and cover 13% and 11% more branches than the direct integration. The worse performance of the direct integration presents the conflicts between these optimizations. In this situation, the conflicts are mainly caused by different criteria definitions. The selection optimized fuzzer selects seeds based on new blocks hit while the other only mutates seeds which cover rare branches. The selected seeds may not meet the criteria definitions of the mutation optimized fuzzer. As a result, the mutation has to keep waiting for its suitable seeds. Moreover, the selection prioritized version and the mutation prioritized version trigger 15 and 18 crashes in openssl and c-areas. In contrast, the direct integration only triggers 5 crashes in openssl. The results demonstrate that the `InteFuzz`-based conflict-aware integration of block-based selection optimization and branch-based mutation optimization improves the fuzzing performance.

4) ***BlBl Group*** *– block-based selection and block-based mutation:* This group represents the type which integrates optimizations of block-based selection and block-based mutation.

From the last three columns of the tables, we find that the performance of the direct integration **Direct-BlBl** degrades, and the selection prioritized version **InteFuzz-BlBlS** as well as the mutation prioritized version **InteFuzz-BlBlM** outperform others. In detail, the selection prioritized version and mutation prioritized version execute 13% and 17% more paths and cover 14% and 14% more branches than the direct integration. The worse performance of the direct integration shows the existence of conflicts. In this situation, the conflicts are mainly caused by different criteria definitions. The selection optimized fuzzer selects seeds based on new blocks hit but the selected seeds may not meet the criteria definitions of the other fuzzer on total block hit count. Furthermore, for the number of crashes triggered, the selection prioritized version and mutation prioritized version trigger 13 and 19 crashes respectively, while the direct integration only triggers 2 crashes. The results illustrate that `InteFuzz`-based conflict-aware integration of block-based selection optimization and block-based mutation optimization improves the performance.

Conflict-aware Integration Conclusion. From the above statistics, we conclude that (1) For the block-based and branch-based optimizations in the seed selection stage and seed mutation stage, their direct integrations would degrade in path and branch coverage. Sometimes it may trigger more unique crashes, but cannot detect more new bugs than individual optimization alone; (2) For any combination of the block-based and branch-based optimizations in the seed selection stage and the seed mutation stage, the `InteFuzz`-based conflict-aware integrations obtain considerable improvements in path and branch coverage. Sometimes it may not trigger more unique crashes, but can always detect more new bugs than individual optimization alone and their direct optimizations integration.

B. GitHub Vulnerabilities Mining

We employ `InteFuzz`-based integration of AFLFast and FairFuzz to fuzz more real-world programs from GitHub, and they also perform well. Although some of these programs, such as libwav and libpng, have been well fuzzed, we still discover 33 unknown vulnerabilities. Among them, 12 are successfully registered as CVEs, as shown in Table IV.

TABLE IV
THE VULNERABILITIES DETECTED BY **INTEFUZZ-BRBRM**

Project	Unknown Vulnera-bilities	CVE-Number or Vulnerability type
Bento4	5	CVE-2018-14531, CVE-2018-14532
libwav	1	CVE-2018-14549
pdf2json	2	CVE-2018-14946, CVE-2018-14947
sound	1	CVE-2018-14948
imageworsener	1	CVE-2018-16782
dbf2txt	1	CVE-2018-17042
doc2txt	2	CVE-2018-17043
simdcomp	1	CVE-2018-17427
pbc	4	CVE-2018-12915, CVE-2018-12917
libpng	1	buffer overflow
thunlp/NRE	2	segment fault
thunlp/Fast-TransX	3	segment fault
tinyrenderer	5	segment fault
pdfalto	4	segment fault, floating Point Exception, infinite loop

Let us take libpng as an example. libpng is an official PNG reference library for reading and writing PNG image files, which has been used and extensively tested for many years. However, `InteFuzz`-based integration still detects one new vulnerability, while AFLFast and FairFuzz do not detect any vulnerability for fuzzing several times. The vulnerability allows remote attackers to cause the buffer overflow via a crafted input. As Listing 3 shows, the vulnerability locates in the function get_token in $pnm2png.c$. The index i in function get_token might be out of range and buffer overflow happens.

We collect the number of paths and branches for fuzzing libpng in 24 hours with AFLFast, FairFuzz, their direct integration, **InteFuzz-BrBrS** and **InteFuzz-BrBrM** in Figure 6

```
void get_token(FILE *pnm_file, char *token)
  {
  ...
  // read string
  do
  {
    ret = fgetc(pnm_file);
    if (ret == EOF) break;
    i++;
    token[i] = (unsigned char) ret; //<==
  buffer overflow occurs here
  }
  while ((token[i] != '\n') && (token[i] !=
  '\r') && (token[i] != ' '));
  ...
}

BOOL pnm2png (...) {
  ...
  char type_token[16];
  get_token(pnm_file, type_token);
  ...
}
```

Listing 3. A vulnerability in libpng which causes buffer overflow.

and Figure 7. It demonstrates that both selection and mutation prioritized version of InteFuzz perform better than others. Although the mutation prioritized version is a little slow in the very beginning due to the more complex initialization, it quickly catches up and surpasses others. At last, two versions of InteFuzz execute more paths and cover more branches than AFLFast and FairFuzz. In contrast, the direct integration performs a little worse than both AFLFast and FairFuzz. Not just on libpng, InteFuzz also performs well on other real-world programs. These practices and comparisons show that InteFuzz successfully benefits from individual optimizations and improves the fuzzing performance through the conflict-aware integration.

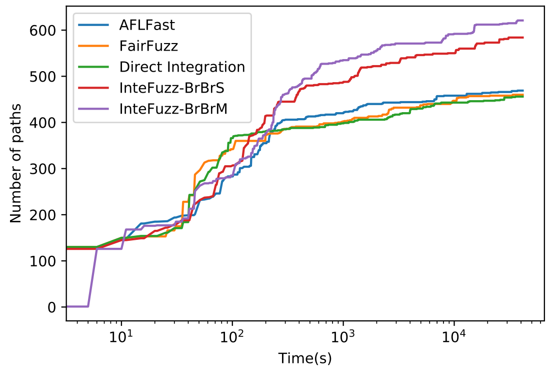


Fig. 6. Number of paths over time of AFLFast, FairFuzz, their direct integration, **InteFuzz-BrBrS** and **InteFuzz-BrBrM** for fuzzing libpng.

VII. Lessons Learned

During the development and practice of InteFuzz in real projects, we get the valuable lessons as follows:

(1) **Even those optimizations have aligned objective and work on different stages, they might have conflicts and directly integrating those optimizations may offset the gains.** Discovering more vulnerabilities and exercising more branches are the aligned objective of most optimizations. On this basis, integrating optimizations of different stages to benefit most from them becomes meaningful. The input seed selection and mutation stage associates with each other closely, and the individual optimizations of them might have different criteria definitions and granularity of runtime information. Thus conflicts are more likely to appear and direct integrations may offset the gains brought by each individual optimization.

(2) **Conflict-aware integration of existing optimizations would greatly improve the performance of fuzzing.** Utilizing feedback information to optimize fuzzing is widely used. However, different feedback information is used in optimizations. To integrate them, a framework which collects multiple feedback information is needed. Assigning priority is a common way to resolve conflicts between processes in the operating system. It also goes for resolving conflicts between optimizations of fuzzing. When there are no conflicts, the overall performance benefits most from each optimization. When conflicts emerge, the overall process could benefit from the optimization with higher priority.

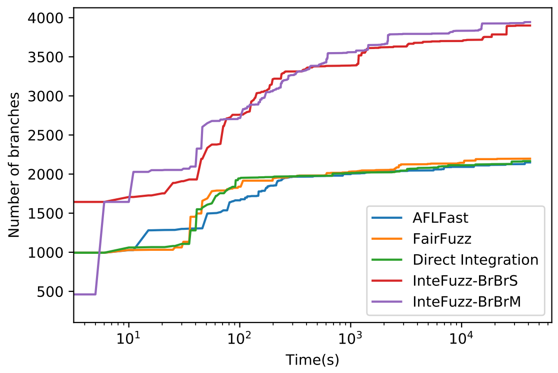


Fig. 7. Number of branches over time of AFLFast, FairFuzz, their direct integration, **InteFuzz-BrBrS** and **InteFuzz-BrBrM** for fuzzing libpng.

VIII. Conclusion

Many optimizations of fuzzing exist but few studies explore the feasibility of integrating multiple optimizations to obtain bigger gains. In this paper, we propose InteFuzz, the first framework that synergically integrates multiple fuzzing optimizations in multiple stages to maximize performance gains. We extract four types of typical optimizations from popular fuzzers and construct direct integrations and InteFuzz-based conflict-aware integrations. In the experiments of fuzzing 8 real-world programs from common Google fuzzer-test-suite benchmark, results show that InteFuzz-based integrations cover more paths and branches and trigger more crashes than direct integrations. We also utilize InteFuzz to fuzz other widely used projects from GitHub. It detects more bugs and 12 of them are successfully registered as CVEs. Our future work will focus on developing dynamic priority assigning mechanism to maximize performance gains and supporting those minor optimizations with special interest.

References

[1] Microsoft security risk detection (”project springfield”). https://www.microsoft.com/en-us/research/project/project-springfield/, 2015. [Online; accessed 26-January-2018].

[2] Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. [Online; accessed 26-January-2018].

[3] Fuzzer test suite. https://opensource.google.com/projects/fuzzer-test-suite, 2016. [Online; accessed 19-September-2018].

[4] Google. honggfuzz. http://honggfuzz.com/, 2016.

[5] Oss-fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017. [Online; accessed 19-September-2018].

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)*, 2017.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[8] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1967–1983, 2019.

[9] Kamal Dahbur, Bassil Mohammad, and Ahmad Bisher Tarakji. A survey of risks, threats and vulnerabilities in cloud computing. In *Proceedings of the 2011 International conference on intelligent semantic Web-services and applications*, page 12. ACM, 2011.

[10] William Drozd and Michael D Wagner. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490*, 2018.

[11] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114. ACM, 2019.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *CollAFL: Path Sensitive Fuzzing*, page 0. IEEE.

[13] Dimitris Geneiatakis, Tasos Dagiuklas, Georgios Kambourakis, Costas Lambrinoudakis, Stefanos Gritzalis, Karlovassi Sven Ehlert, and Dorgham Sisalem. Survey of security vulnerabilities in session initiation protocol. *IEEE Communications Surveys & Tutorials*, 8(3):68–81, 2006.

[14] Sam Hocevar. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 26-January-2018].

[15] Rahul Johari and Pankaj Sharma. A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection. In *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*, pages 453–458. IEEE, 2012.

[16] T Kavitha and D Sridharan. Security vulnerabilities in wireless sensor networks: A survey. *Journal of information Assurance and Security*, 5(1):31–44, 2010.

[17] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan. A survey on sql injection: Vulnerabilities, attacks, and prevention techniques. In *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, pages 468–471. IEEE, 2011.

[18] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[19] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–814. ACM, 2018.

[20] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 562–566. IEEE, 2018.

[21] Guei-Yuan Lueh. Static compilation of instrumentation code for debugging support, November 15 2005. US Patent 6,966,057.

[22] Zhengxiong Luo, Feilong Zuo, Yu Jiang, and Jian Gao. Polar : Function code aware fuzz testing of ics protocol. In *2019 International Conference on Embedded Software (EMSOFT)*. IEEE, 2019.

[23] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[24] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[26] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[27] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 986–995. ACM, 2019.

[28] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[29] Mingzhe Wang, Liang Jie, Yuanliang Chen, Yu Jiang, Jiao Xun, Liu Han, Zhao Xibin, and Sun Jiaguang. Safl: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Software Engineering Companion (ICSE-C), 2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 2018.

[30] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328. ACM, 2017.

[31] Michal Zalewski. American fuzzy lop, 2015.