DeepFuzzer: Accelerated Deep Greybox Fuzzing

Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao,
Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo

Abstract—Fuzzing is one of the most effective vulnerability detection techniques, widely used in practice. However, the performance of fuzzers may be limited by their inability to pass complicated checks, inappropriate mutation frequency, arbitrary mutation strategy, or the variability of the environment.

In this paper, we present DeepFuzzer, an enhanced greybox fuzzer with qualified seed generation, balanced seed selection, and hybrid seed mutation. First, we use symbolic execution in a lightweight approach to generate qualified initial seeds which then guide the fuzzer through complex checks. Second, we apply a statistical seed selection algorithm to balance the mutation frequency between different seeds. Further, we develop a hybrid mutation strategy. The random and restricted mutation strategies are combined to maintain a dynamic balance between global exploration and deep search.

We evaluate DeepFuzzer on the widely used benchmark Google fuzzer-test-suite which consists of real-world programs. Compared with AFL, AFLFast, FairFuzz, QSYM, and MOPT in the 24-hour experiment, DeepFuzzer discovers 30%, 240%, 102%, 147%, and 257% more unique crashes, executes 40%, 36%, 36%, 98%, and 15% more paths, and covers 37%, 34%, 34%, 101%, and 11% more branches, respectively. Furthermore, we present the practice of fuzzing a message middleware from Huawei with DeepFuzzer, and 9 new vulnerabilities are reported.

Index Terms—Software Testing, Greybox Fuzzing

1 Introduction

Fuzzing is a software testing technique that feeds the computer program with invalid, unexpected or random input seeds with the goal of monitoring exceptions. The first (or at least best-known) fuzzing project aimed to test the reliability of UNIX utilities [1], [2]. Since then, fuzzing has identified a large number of software bugs and security vulnerabilities. At present, fuzzing is widely deployed by many companies such as Google [3], [4], Microsoft [5], [6], and Adobe [7].

To generate input seeds, mutation and generation are two popular methods [1]. Mutation-based fuzzers mutate existing seeds randomly so as to construct new seeds [8], [3]. Aimless random mutation produces a lot of worthless seeds, especially for complex programs. Because of this, they teeter in shallow areas and difficult to reach deep places in programs. However, many serious vulnerabilities usually remain undiscovered. Generation-based fuzzers, on the other hand, normally test programs accepting highly-structured inputs (such as SQL and JavaScript) [9], [10]. While a significant amount of up-front work is required to study specifications and construct rules, semantic checks like parsing are still difficult to pass. The limited generation rules also restrict the vitality of inputs to explore the state space of the program.

To strike a balance between effectiveness and automation, many researchers have devoted significant efforts. One typical solution is guiding fuzzing with coverage information. American Fuzzy Lop (or simply AFL) [8] is one of the representative guided fuzzers. With some manual-provided initial seeds, it explores the target program by mutating them and retaining interesting seeds which discover new coverage. Then it repeatedly mutates interesting seeds to cover more states of the program. In this way, it has hunted hundreds of high-impact vulnerabilities. However, just adding coverage information is insufficient for exploring hard-to-reach regions in programs and thus misses some deep vulnerabilities. First, AFL does not provide qualified initial seeds for initial start points even as initial start points are crucial; extra cycles of mutation on low-quality initial seeds hardly improves the performance. Klees et al. also point out the impact of initial seeds to fuzzing [11]. Second, while worthless seeds can be screened out with coverage information, the blind mutation strategy does not know how to mutate the meaningful seeds to reach hard-to-reach regions. To get through complex checks and discover these regions, not only is a large number of mutations needed, but also a bit of luck. In addition, when AFL gets through complex checks and covers hard-to-reach regions with a lot of efforts, the byte sequence related to these checks are fragile in the presence of the blind mutation strategy.

- *Jie Liang, Yu Jiang, Mingzhe Wang, and Yuanliang Chen are with the School of Software, Tsinghua University, China, also with Beijing National Research Center for Information Science and Technology, China, and also with Key Laboratory for Information System Security, Ministry of Education, China.*
- *Xun Jiao is with the Department of Electrical and Computer Engineering, Villanova University, USA.*
- *Houbing Song is with the Department of Electrical, Computer, Software, and Systems Engineering, Embry-Riddle Aeronautical University, USA.*
- *Kim-Kwang Raymond Choo is with the Department of Information Systems and Cyber Security, University of Texas at San Antonio, USA.*

Manuscript received Jul. 23, 2019; revised Nov. 11, 2019; accepted Dec. 6, 2019. This research is sponsored in part by the NSFC Program (No. 61527812), the National Science and Technology Major Project of China (No. 2016ZX01038101). Corresponding author: Yu Jiang (e-mail: jy1989@mail.tsinghua.edu.cn).

Researchers have extended AFL in various ways. Some optimize the mutation strategy to acquire better efficiency. For example, AFLFast [12] assigns more mutation times to seeds which execute infrequency paths, but this does not increase the possibility of reaching uncovered regions. FairFuzz [13] adjusts the mutation strategy to direct fuzzing to rare regions, but its performance is vulnerable to local convergence. AFLGo [14] assigns more mutation times to seeds closer to target regions, so as to direct the fuzzing to specific locations. However, any enhancement from reaching deep regions is slight. MOPT [15] utilizes a customized particle swarm optimization algorithm to find the optimal selection probability distribution of mutation operators to improve fuzzing.

Others try to incorporate program analysis techniques such as symbolic execution as a way to extend AFL. For example, SAFL [16], Driller [17], and QSYM [18] all successfully apply symbolic execution to solve complex checks which trap AFL. But solving complex constraints is still difficult for state-of-the-art constraint solvers, and AFL is likely to get stuck again quickly because of the blind mutation strategy.

Last but not least, those academic extensions perform well when evaluated in ideal environments, but they do not work consistently as expected when applied to many industrial projects. Due to the complexity and variability of the production environment, there are too many potential causes that could fail a "smart" fuzzer. For instance, two testing engineers from Huawei tried to fuzz their industrial message middleware `libmsg` with Driller, but they gave up after investing two weeks of valuable time. They discovered that Driller lacks the support for a set of POSIX APIs and has rigid requirements on system and library configuration.

Based on the performance of existing fuzzers, we observe that there are four major obstacles for implementing effective and adaptable fuzzing.

1) **Poor initial seed quality.** Initial seeds supply start points to get through complex checks. Efficient seeds are hard to manually construct and random seeds are worthless. Thus, generating guiding initial seeds automatically is badly needed.
2) **Unbalanced seed selection.** Unbalanced seed selection may assign seeds with inappropriate mutation times. Valuable seeds are seldom selected, which leads to inadequate mutations. In contrast, some seeds are selected too much which leads to a waste of resources.
3) **Aimless mutation strategy.** Extra efficiency gained by coverage information enables exploring more meaningful areas, but aimless mutation strategies invalidate this benefit. These strategies mutate inputs randomly so that the structures required to pass complex checks are easily damaged. Once the delicate structures are broken, the exploration can reach only the shallow regions.
4) **Complex industrial environment.** Fuzzers cannot easily manage the complexity and diversity of real industrial environments. Better adaptability of fuzzers to industrial software products is much needed.

In this paper, we present DeepFuzzer[1], a fuzzer that combines qualified seed generation, balanced seed selection, hybrid seed mutation, and automatic fuzzing environment configuration so as to grapple effectively with the four obstacles of current fuzzing techniques, and explore deep paths as widely and as fast as possible. The key idea is that the seed that touches the hard-to-reach regions represents good directions and should be automatically generated, selected more and mutated more times, and the mutation on these seeds are restricted to some positions and operations to reserve the path depth and breadth. Specially, we employ symbolic execution [19] in a lightweight approach to generate high quality initial seeds which pass complex checks. Then, we apply a statistical seed selection algorithm to balance the mutation frequency of different categories of seeds. We also develop a hybrid random and reachability-reserved mutation strategy that decides how to mutate the selected seeds depending on the rarity of the reached regions. This achieves a dynamic balance between global exploration and a deep search. Furthermore, to enhance the adaptability of DeepFuzzer to complex industrial environments and system building complexity, we develop an automatic environment configuration component named toolchain.

DeepFuzzer is evaluated on the widely used benchmark Google fuzzer-test-suite[2] which consists of real-world programs. The number of paths executed, branches covered, and unique crashes triggered are used as metrics. Compared with the state-of-the-art fuzzers such as AFL, AFLFast, FairFuzz, QSYM, and MOPT used in the 24-hour experiment, DeepFuzzer performs better, triggering 30%, 240%, 102%, 147%, and 257% more unique crashes, executing 40%, 36%, 36%, 98%, and 15% more paths and covering 37%, 34%, 34%, 101%, and 11% more branches, respectively. Furthermore, we present the practice of fuzzing an industrial message middleware from Huawei with DeepFuzzer. After epitomizing several typical but interesting industrial environment traps, 9 previously-unknown vulnerabilities are reported, including flaws that lead to denial of service or communication crashes. The main contributions of our study are as follows:

- We propose DeepFuzzer, an efficient open source fuzzer with qualified seed generation, balanced seed selection, and hybrid seed mutation.
- We list typical obstacles encountered while applying fuzzing to a real industrial project: `libmsg` from Huawei. Solutions to each obstacle are also provided.

This paper is organized as follows: Section 2 introduces a motivating example. Section 3 details our approaches with DeepFuzzer to accelerate deep greybox fuzzing. Section 4 presents design and implementation details. Section 5 presents evaluation along with obstacles and solutions for industrial practice. Section 6 introduces the related work. We conclude by exploring implications in section 7.

2 Motivating Example

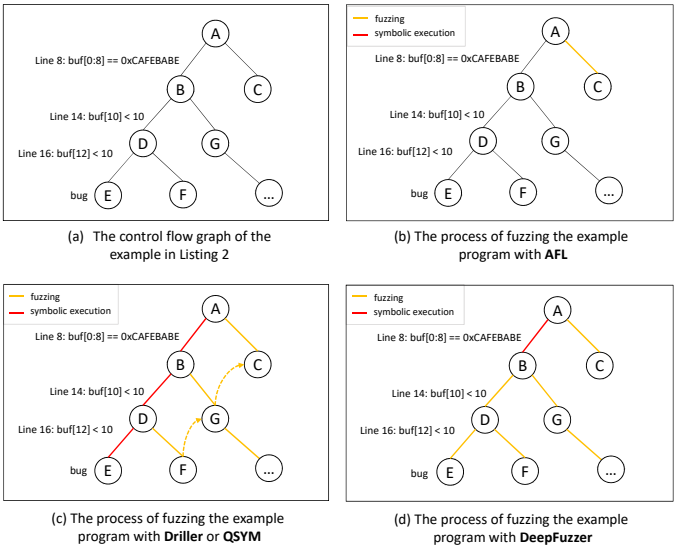
To show how DeepFuzzer can address complicated checks, we walk through a code snippet presented in Listing 1. In

1. https://github.com/Ljiee/deepfuzz
2. https://github.com/google/fuzzer-test-suite/

```
1  int fun(char *filename, int size) {
2    unsigned char buf[1000]; int fd;
3    if((fd = open(filename, O_RDONLY))==-1)
4      exit(0);
5    read(fd, buf, size);
6    if(size > 1000) return -1;
7    //BLOCK A
8    if(*(uint64_t *)buf == 0xCAFEBABE)
9      //BLOCK B
10     printf("Magic bytes!\n");
11   else
12     //BLOCK C
13     return -1;
14   if(buf[10] < 10) {
15     //BLOCK D
16     if(buf[12] < 10) {
17       //BLOCK E
18       bugs();
19     } else {
20       // BLOCK F
21       ...some other task...
22     }
23   } else {
24     //BLOCK G
25     ...some other task...
26      close(fd); return 0;
27   }
28   close(fd); return 0;
29 }
```

Listing 1. A motivating example illustrates that how DeepFuzzer addresses complicated checks.

this code snippet, the paths covered depend on certain bytes at fixed offsets. When fuzzing this program, AFL will be stuck in Block C; Driller or QSYM will cost a lot of time and resources to trigger the bug; DeepFuzzer will trigger the bug easily and quickly. To clarify such differences, the corresponding control flow and the testing process of AFL, Driller or QSYM, and DeepFuzzer are presented in Figure 1.



(a) The control flow graph of the example in Listing 2

(b) The process of fuzzing the example program with **AFL**

(c) The process of fuzzing the example program with **Driller** or **QSYM**

(d) The process of fuzzing the example program with **DeepFuzzer**

Fig. 1. The control flow graph of Listing 2 and the process of fuzzing the program with AFL, Driller / QSYM, and DeepFuzzer.

In line 8 of Listing 1, buf[0] and buf[1] are compared with certain bytes to validate the input. This pattern is common in real-world programs such as libjpeg, lcms, and libpng. However, it is a significant obstacle for pure AFL-family fuzzers (AFL, AFLFast, and FairFuzz). Figure 1 (b) shows the operation of AFL. Because it has no whitebox-level knowledge about the program, it will get stuck at Block C and guess with brute-force by mutating different bytes on various offsets.

Figure 1 (c) shows the process of Driller or QSYM. They cannot pass the check in line 8 directly, but symbolic execution can help them to arrive at Block B. However, they ignore the importance of bytes from 0 to 8 of buf. Their following mutated seeds generated by fuzzing might destroy these bytes and return back to Block C again. As a result, they have to employ symbolic execution to pass line 14 and get to Block D. Note that the check at line 14 is not very difficult, so solving it by symbolic execution is a waste of resources to some extent. The check at line 14 also causes more input seeds covering Block G than Block D. So Driller or QSYM takes more times to fuzz seeds who hit Block G and most mutated seeds of them would go back to Block C. It is difficult for them to concentrate on fuzzing Block D and to cover its descendant blocks. In the end, they might have to use symbolic execution again to reach Block E and trigger the bug. On the other hand, along with the increased depth of the path, the cumulative constraints are also increasing. So Driller or QSYM will spend more and more time to solve these nested conditions.

Figure 1 (d) shows the operation of DeepFuzzer. It passes the check at line 8 and gets to Block B at the beginning owing to qualified initial seeds generated by symbolic execution. Next, DeepFuzzer detects that offset 0 to 8 are crucial to Block B. It preserves them and mutates other bytes to get Block D. DeepFuzzer utilizes a balanced seed selection, which means it balances Block D and G by giving the seeds who hit Block D more mutation times. On top of that, its restrict mutation preserves bytes in offset 0-8, 10, 12 and only mutates other offsets. As a result, DeepFuzzer finds Block E and triggers the bug quickly.

When testing real-world programs, the whole process of Driller or QSYM would cost more time and resources. The inefficient use of the results of symbolic execution causes frequent switching between symbolic execution with fuzzing. The long execution trace would accumulate constraints thus lower the success rate in solving. In contrast, DeepFuzzer combines symbolic execution in a more suitable way for fuzzing real-world programs. In summary, DeepFuzzer is superior to other fuzzers in the following three aspects:

1) DeepFuzzer employs symbolic execution to generate initial seeds, which ensures that the complicated checks could be passed in the beginning. It also preserves the efficient fuzzing process and avoids the switching cost between the symbolic execution and fuzzing.
2) DeepFuzzer utilizes the balanced seed selection to give adequate mutation times for each seed. It avoids wasting too many resources on exploring frequent parts.
3) DeepFuzzer integrates the restricted and random mutation to retain the results of the symbolic execution. As a result, DeepFuzzer passes the nested conditions more easily and explores more and deeper paths continually.

3 DeepFuzzer Approach

Figure 2 presents an overview of the proposed approach. Seed generation takes the C/C++ source code as input and utilizes symbolic execution to generate qualified initial seeds to form the original circular queue. During the fuzzing process, seeds are taken out from the queue one after another. Seed selection determines whether to skip a seed and compute the mutation times for each selected seed. Seed mutation is responsible for mutating seeds in the restricted or random strategy based on whether the seed touches hard-to-reach areas. The bug report is produced by the seeds which crash the program. Or if the mutated seeds have new coverage, they will be added to the seed queue for further mutations. Otherwise, the worthless mutated seeds will be thrown into the trash. The process of AFL could be contained in DeepFuzzer, which is framed in the dotted box. There are three main differences between AFL and DeepFuzzer. First, AFL lacks the process of seed generation. Its initial seeds rely on manual supply. Second, it selects seeds without considering the branch count statistics. Last, it mutates seeds only by random mutation.

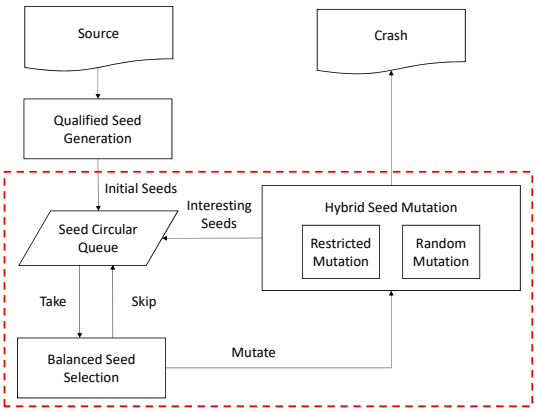


Fig. 2. The interaction of DeepFuzzer algorithms.

3.1 Qualified Seed Generation

If a complex check can not be passed by initial seeds, it takes significant computing resources to generate satisfied seeds by pure luck. To compensate the format-agnostic nature of greybox fuzzing, a symbolic executor is used to produce high-quality initial seeds.

The symbolic executor works online and forks on branches. To model the interaction between the program and the environment, a subset of POSIX API is simulated symbolically. For faster collection of constraints, LLVM bitcode is chosen as the program representation to reduce the instruction count. Different from normal whitebox fuzzers such as Driller, the symbolic executor uses a strict time-limited scheduler, because once the code can be covered by initial seeds, fuzzer is more efficient at triggering bugs. Following the design principle that a symbolic executor should be lightweight, the total execution time, maximum memory usage, and constraint solving time are strictly limited to prevent wasting computing resources on unnecessary explorations.

```
1  //BLOCK A
2  if(x == 0x12345678) {
3  //BLOCK B
4      if(...) {
5      //BLOCK D
6      }
7  }
8  else {
9  //BLOCK C
10     if(...) {
11     //BLOCK E
12     }
13 }
```

Listing 2. Unfair selection.

When the symbolic executor halts, states are filtered to only export seeds covering new paths. This optimization is crucial, because one symbolic execution session might contain tens of thousands of states, many of which exercise the same path. The optimization drastically reduces the number of seeds passed to the fuzzer.

3.2 Balanced Seed Selection

If valuable seeds are seldom selected, it would lead to inadequate mutations. In contrast, if some worthless seeds are selected too frequent, it would waste resources. The balanced seed selection aims at assigning balanced mutation times to each seed. We propose two rules named the branch-sensitive fair skip rule and the branch-sensitive energy assignment rule to make the fuzzing tend to be in equilibrium.

Branch-sensitive fair skip rule. In AFL, whether a seed is worth fuzzing depends on three factors: the size, the running speed, and the number of fuzzed times. The skip rule is fair for seeds, as the seed characteristics are the main consideration. However, it is not fair for branches, as the branch coverage distribution is left out. For each branch, AFL only favours the smallest and the fastest seed covering the branch. In AFL, if a seed is not favored or it has been fuzzed before, AFL will skip it, which may lead to coverage misses of some critical branches.

For example, consider the code snippet in Listing 2. We use tuples AB, AC, BD, CE to represent branches. Suppose the seed set S_1 and S_2 contain seeds which can hit branch AB and AC separately. Intuitively, if we select each seed the same number of times, the probability of generating seeds covering CE is much bigger than BD. Branch AB is blocked by the magic bytes in a condition statement, leaving explosive growth in the size of S_2. So for the branch BD and CE, the selection is not fair. To fix this problem, we should give a high priority to seeds in S_1 and skip seeds in S_2 with a certain probability. In order to do this, we assign a skip probability for every seed. This probability is inversely proportional to the occurrence frequency of the code block.

We collect and use the branch hit count to indicate the occurrence frequency to be common or rare. A branch represents a basic block transition; its hit count reflects the frequency of its associated blocks. Taking all runs into account, the *rarest branch* of a seed denotes the branch which has the minimum hit count among the hit branches of this seed. Let x_i be the hit count of the *rarest branch* for the seed s_i, n be the number of branches. γ is a constant,

which diminishes the skip probability equally to increase efficiency. The optimized fair skip probability for seed s_i is:

$$P(s_i) = \left(1 - \frac{x_i^{-1}}{\sum_{j=1}^{n} x_j^{-1}}\right) \cdot \gamma \quad (1)$$

The formula above assigns a higher mutation probability for seeds exercising infrequent hit branches. After balancing branches, we still follow the original skip rule of AFL to omit unfavoured seeds.

Branch-sensitive energy assignment rule. Energy is defined as the number of times seeds are mutated in the random stage of fuzzing. AFL gives more energy to recently created seeds which spend less execution time and achieve more branch coverage. This heuristic helps AFL reach more regions with less cost but also neglects the rarity of branches, which usually leads to unbalanced energy assignment. For example, a seed which only covers common branches might run fast. Conversely, another seed runs slow but can hit rare branches. Because of the lack of branch rarity, the assigned energy for the first seed is excessive while for the latter is inadequate.

To assign a more suitable number of mutation times to seeds, we take the branch rarity into account similar to the seed selection stage. Let s denote the selected seed that needs to be mutated next, $p(s)$ denote the energy of s, and $p_{AFL}(s)$ denote the original energy calculated by AFL. Given the number of times $c(s)$ which s has previously been chosen from the queue S and the hit number $h(s)$ of the *rarest branch* covered by s, DeepFuzzer computes $p(s)$ as

$$p(s) = \min\left(\frac{p_{AFL}(s)}{\beta} \cdot \frac{2^{c(s)}}{h(s)}, M\right) \quad (2)$$

$p(s)$ is inversely proportional to $h(s)$, which means the rarer branch a seed can cover, the more energy the seed will get. This helps to make the fuzzing process more balanced towards the branch. The increase in $c(s)$ gives the exponential energy for reselected seeds to explore the states. The constant M provides an upper bound on the number of mutations per fuzzing iterations, which is 160k, the same as with AFL. β balances the relation between the original AFL energy assignment value and the branch-sensitive energy assignment value.

3.3 Hybrid Seed Mutation

Before the main fuzzing process starts, seed generation has supplied high-quality initial seeds automatically. These seeds give the fuzzing a lot of valuable directions and help the fuzzer pass the complicated checks (like line 2 in Listing 2) easily. But normal mutation strategy does not know the delicate structure which is required to pass these checks. Random mutation strategies easily damage the structure and mess up the high-quality seeds, such as in Driller and QSYM. Defeated by the complex checks, the fuzzing session has to wander in shallow areas. The useless mutation also wastes too many resources and causes inefficient fuzzing.

In order to take full advantage of the initial seeds and to explore the deep places while maintaining breadth, we propose a hybrid mutation strategy which combines the restricted-mutation strategy and the random mutation strategy. The restricted-mutation strategy mutates seeds which cover tough reachable areas in a restricted way to preserve the depth. For other seeds, DeepFuzzer preserves them to retain the breadth with the random mutation strategy.

After several cycles, the original rare branches degenerate into common branches and new rare branches appear. When it is difficult to cover rare branches, it is still possible to utilize the random mutation strategy to further increase the coverage. As described in the evaluation section, the hybrid mutation solves the local convergence problem to a certain extent, such as in FairFuzz. A branch is rare if its hit count is less than or equal to $rarity$. $rarity$ is the smallest power of two, which bounds the number of input seeds hitting the rarest branch. Let min_hit be the minimum hit count of branches, $rarity$ is:

$$rarity = 2^{\lceil \log_2 min\_hit \rceil}$$

A *target branch* of a seed is the rarest branch among all covered branches of the seed. With the *target branch* chosen, the restricted mutation strategy can be performed. The algorithm finds out strategies which would not diverge from the *target branch*: for each byte in the seed, can it be overwritten, deleted, or inserted? The overwritable bytes are computed by replacing every byte of the input with a random byte. If the result still hits the *target branch*, the byte is designated as overwritable. The property of deletable and insertable is detected in a similar way: delete the byte being tested, or insert a random byte before. Although three operations all change the value of the testing byte, they will cause different external results and all of them are indispensable. Take the JPEG file as an example, it has several marker sections which have fix length and data sections. We can only overwrite bytes in the marker sections because other operations will change the length and cause an invalid format. But in the data sections, we can apply all of the three operations and still get a valid JPEG file. Knowing the mutation property for each byte, the seed is mutated in restricted ways at corresponding positions. It is noteworthy that this algorithm utilizes random bytes for replacement or insertion.

4 Implementation

DeepFuzzer consists of three main components — the seed generator, the fuzzer, and the toolchain. The symbolic seed generator is based on KLEE. The improved fuzzer is based on AFL. The toolchain is based on a standard system toolchain invocation. KLEE and AFL work on vastly different levels of program representation. To help bridge the gap, DeepFuzzer follows the interface specification of libFuzzer. With this layer of abstraction, DeepFuzzer can use different entry points for symbolic execution and fuzzing. This is automatically done by the toolchain component.

The symbolic seed generator is responsible for producing high-quality initial seeds. It is a wrapper of KLEE. The source code of KLEE is not modified, but the search strategy is fine-tuned for seed generation. When the symbolic executor halts, states are filtered to only export seeds covering new paths. Besides the tuning, there are three adaptations. First, to support symbolic execution and basic POSIX API simulation, we ship KLEE built with uclibc. Second, we set the execution time of KLEE to 20 minutes. Finally,

we develop a converter to convert the results of KLEE to available seeds.

The fuzzer component changes the source code of AFL. It modifies the logic of seed selection and seed mutation to implement the algorithm we proposed. The seed selection algorithm applies the skip rule and the energy assignment rule to balance the mutation times of branches. Three maps are defined: one counts the hit number of every branch, one labels whether a branch is interesting, denoting the rarest of all branches a seed can cover. The last one records the times a seed is successfully selected for fuzzing. When a seed is taken out from the queue, we skip it with the probability calculated by the formula (1) in section 3.2. The energy which is computed as the formula (2) in section 4.2 controls the random mutation times for each seed. The hybrid seed mutation strategy assigns each seed with a fittest mutation strategy. The restricted mutation strategy is only used for the seeds that hit rare branches, it finds which bytes in a seed can be overwritten, deleted, or inserted, and the rules are obeyed in the mutation stage.

The toolchain component intercepts standard system toolchain invocation, which enables customization on the flags passed to the compiler and linking stage. Depending on different build modes (symbolic vs fuzzing), different sets of compiler flags are injected. The toolchain component takes care of the trivial details of different build systems such as CMake or AutoConf. It injects the flags as the specification of build systems. When building LLVM bitcode for symbolic execution, DeepFuzzer uses a pre-built bitcode file to supply the `main` function when linking. It feeds its argument(`argv`) as input, which simplifies the environment interaction modelling of KLEE. The linking stage also combines all the LLVM bitcode files into a whole file for KLEE. When building an executable file for fuzzing, a slightly patched AFL driver in libFuzzer is used as the `main` function for linking. The patch adds support for clean-up logic at exit. From our observation, programs with background threads tend to crash because of an incorrect exit procedure; false positives on memory leakage are also found. Moreover, the toolchain also enables sanitizers such as ASAN to detect latent bugs. The toolchain component also adds an additional LLVM pass to collect coverage information.

DeepFuzzer encapsulates the internal complexities above into a collection of command line utilities. The command line utilities forms a workflow for software developers, providing an easy-to-use interface. The utilities detect the system environment and configure each internal tool. In this way, optimal parameters are automatically decided, thus software developers do not have to manually tune the parameters. Furthermore, from our empirical study, one of the major obstacles for complex fuzzers to run on real industry systems is system configuration inconsistency. A compatibility layer is implemented in DeepFuzzer, which contains various software library dependencies of each component. As long as the kernel and glibc is recent enough, DeepFuzzer is able to run on any Linux distribution.

5 Evaluation

We evaluate the effectiveness of DeepFuzzer on the widely used benchmark Google fuzzer-test-suite and other real-world programs. The influence of symbolic seed generation, balanced seed selection, and hybrid seed mutation are quantified. Compared with the state-of-the-art fuzzers, DeepFuzzer performs better. For example, compared with one of the most recently developed symbolic execution aided tool — QSYM, it triggers 147% more unique crashes, covers 98% more paths, and achieves 101% more covered branches respectively. Furthermore, we deployed DeepFuzzer to the highly customized runtime environment of Huawei and fuzzed the message middleware `libmsg`. After epitomizing several typical but interesting industrial environment traps, 9 previously-unknown vulnerabilities were reported, including flaws that lead to denial of service or communication crash. On this basis, we summarize ways to improve the utility of fuzzers in the industrial practical environment.

5.1 Benchmark Evaluation

5.1.1 Experiment setup

Google fuzzer-test-suite is a widely used fuzzing benchmark, including real-world projects such as boringssl, c-ares, guetzli, lcms, libarchive, libssh, libxml2, pcre2, proj4, and re2. They are derived from real-life libraries which have typical bugs, hard-to-find code paths and other challenges for bug-finding tools. We use three metrics to evaluate the results of fuzzers on these ten real-life projects. These metrics consist of the number of exercised paths, covered branches, and triggered unique crashes. The first two metrics evaluate the coverage of the target programs. AFL distinguishes the crash by different execution paths. Although several unique crashes which might point to the same bug, no bugs can be found without a crash. In other words, the discovery of unique crashes is a prerequisite for the discovery of bugs. Thus this number reveals the probability of finding vulnerabilities.

To quantify the effect of seed generation, balanced seed selection, and hybrid seed mutation, respectively, we separately integrate each component on the original implementation of AFL. Then we compare DeepFuzzer with some AFL-family tools consists of AFL, AFLFast, and FairFuzz. Two state-of-the-art fuzzers namely QSYM and MOPT are also compared with DeepFuzzer. In particular, QSYM is the most recently developed symbolic execution aided fuzzing tool. Because QSYM needs at least two cores (it requires an AFL instance to run simultaneously), we also run DeepFuzzer and MOPT with two cores. In addition, AFL, AFLFast, and FairFuzz are also compared in the same environment settings. To have a better comparison, except for Google fuzzer-test-suite, we also evaluate these fuzzers on three large real-world programs namely ffmpeg, openjpeg, tcpdump. Apart from the three metrics namely paths executed, branch covered and crash triggered, we also analyze the bugs found by each fuzzer.

We run each tool accompanied with Google AddressSanitizer (ASAN) for 24 hours on a 64-bit machine which has 32 cores (Intel(R) Xeon(R) CPU E5- 2630 v3 @ 2.40GHz), 128GB of main memory, and uses Ubuntu 16.04 as host OS.

5.1.2 Is qualified seed generation positive or negative

We first verify whether the seed generation mechanism can generate high-quality seeds to optimize the performance of fuzzing or not. We feed AFL with initial seeds generated by our symbolic-execution-based generator. Synthetically considering the time to generate enough high-quality initial seeds and the fuzzing performance, we choose 20 minutes for symbolic execution. The average count of initial input seeds generated by symbolic execution is 70. Table 1 shows the total number of three metrics for the benchmark in 24 hours. For comparison, we also list the statistics of original AFL. To have a fair comparison, we generate the same number of random valid inputs for AFL.

We observe that the initial seeds increase 17% more paths executed and 23% more branches covered. Benefiting from increased coverage, the improved version finds about 150% more unique crashes than running AFL.

TABLE 1
Influence on Qualified Seed Generation

Metric	AFL + Random Seed Generation	AFL + Qualified Seed Generation
Path	22522	26249
Branch	63371	77907
Crash	244	610

The improvements in performance have two reasons. First, symbolic execution generates seeds which can pass some complicated checks. Many paths and vulnerabilities can only be found after passing these complicated checks. However, it is very difficult to produce these seeds by mutation alone. Second, these seeds provide plenty of directions for fuzzing and widen the breadth of the fuzzing process. Without these seeds, AFL may get trapped in a few easily discovered directions, which limits it to finding new paths in the later stage. As the coverage increases, the probability of finding vulnerabilities also increases.

Let us take the detail of fuzzing pcre2 from the benchmark as an example, as presented in Figure 4. The number of paths and branches are higher than with random seeds almost from the beginning to the end. High-quality initial seeds bring the benefits of discovering more paths and branches to the improved version at the very beginning. With the help of qualified initial seeds, AFL executes deeper paths. The abundant directions and the ability to easily reach deep regions increase the coverage steadily and evenly. In contrast, when feeding AFL with random initial seeds that have scarce directions, its speed to find new regions is low. Profiting from the increased coverage, the improved version triggers more unique crashes in the long run.

We also analyze the branch coverage when fuzzing pcre2 with initial seeds generated by random or symbolic execution. As Figure 3 shows, their branch coverage is not same. They both have their own unique branches but AFL finds about 3000 more unique branches when runs with qualified initial seeds.

Qualified seed generation produces a set of high-quality initial seeds which exercise deep paths. Those seeds are exclusive to the version with qualified initial seeds, as complex

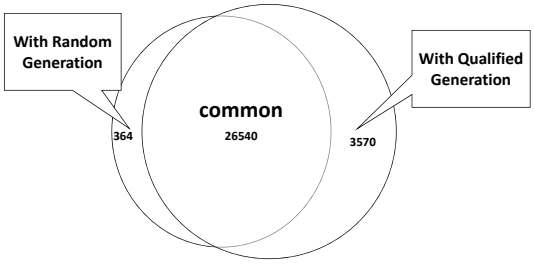


Fig. 3. The Venn diagram of branch coverage with random or qualified seed generation for fuzzing pcre2.

checks in deep paths cripple traditional greybox fuzzing. With the exclusive seeds, this version first achieves higher coverage, which enables unique opportunities to explore state spaces behind those deep paths. Thus many crashes hidden in these paths are only triggered when fuzzing with seed generation. On the other hand, when running AFL with qualified initial seeds, it executes more paths in a shorter time. That means AFL could try more branch combinations and as a result, trigger more crashes.

From these statistics and analysis, we find that the symbolic-execution-based seed generation is positive to fuzzing. It not only improves the coverage but also boosts the probability of finding vulnerabilities.

5.1.3 Is balanced seed selection positive or negative

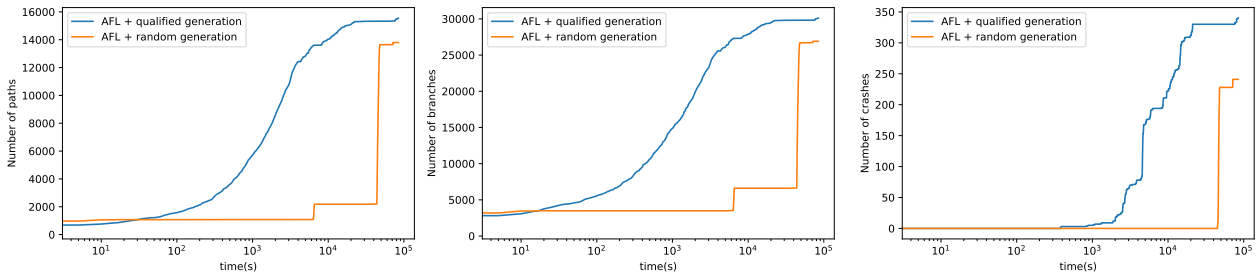
We implement the balanced seed selection algorithm on AFL and verify whether it works or not. The result presents that it assists AFL in growing both the number of paths and branches for each tested project. As Table 2 shows, the increase in the number of paths and branches are 12% and 14% respectively. And in total, it triggers 122% more unique crashes.

TABLE 2
Influence on Balanced Seed Selection

Metric	AFL	AFL + Balanced Seed Selection
Path	20279	22656
Branch	61974	70559
Crash	211	469

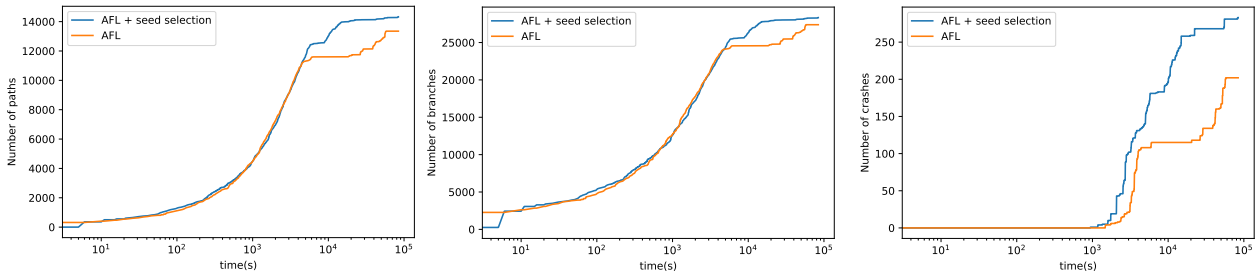
The main reason for the performance improvement is that the balanced selection treats each branch equally, giving the corresponding seed a suitable number of mutation times so that each seed can generate enough meaningful results without wasting resources. In contrast, AFL does not take the rarity of branches into account, which limits the effectiveness. Some worthless seeds are mutated too many times while others are not mutated enough. As a result, in the same amount of time, we find more branches and paths. Accordingly, the probability of finding vulnerabilities also increases.

Figure 5 shows the performance comparison of fuzzing pcre2. In the beginning, there are no marked differences between them. However, because of the more balanced choice and the reasonable energy given to seeds, all metrics continue to rise while AFL cannot find new paths any more. With the increase of coverage, it also triggers more crashes.



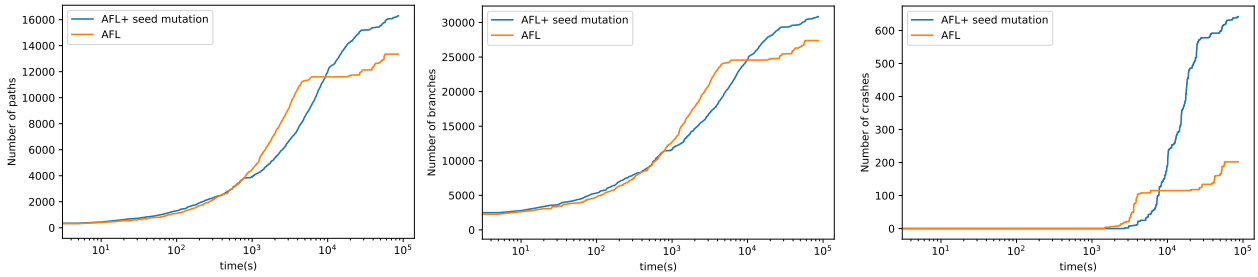
(a) paths over time (b) branches over time (c) crashes over time

Fig. 4. Number of paths, branches, and unique crashes change over time for fuzzing pcre2 with seed generation.



(a) paths over time (b) branches over time (c) crashes over time

Fig. 5. Number of paths, branches, and unique crashes change over time for fuzzing pcre2 with seed selection.



(a) paths over time (b) branches over time (c) crashes over time

Fig. 6. Number of paths, branches, and unique crashes change over time for fuzzing pcre2 with seed mutation.

From these statistics and analysis, we conclude that the balanced seed selection is positive to fuzzing. It helps AFL cover more states steadily and hunt more bugs.

5.1.4 *Is hybrid seed mutation positive or negative*

We implement the hybrid mutation strategy on AFL. As Table 3 demonstrates, the total number of covered paths increases 24%, the total number of branches covered increases 18%, and the total number of triggered unique crashes grows 123%.

TABLE 3
Influence on Hybrid Seed Mutation

Metric	AFL	AFL + Hybrid Seed Mutation
Path	20279	25215
Branch	61974	73397
Crash	211	471

The main reason for the performance improvement is that the hybrid seed mutation combines the branch-reserved mutation to keep depth and the random mutation to keep breadth. As for the seeds which can cover rare branches, the branch-reserved mutation makes it possible to explore the deeper paths which still contain the original rare branches. For the other seeds, the random mutation strategy is applied thus other common branches can be found with more probability.

We also use pcre2 to demonstrate more details. Figure 6 indicates that the growth rate of the coverage (the number of paths and branches) are almost the same at the beginning. Then AFL gradually exceeds DeepFuzzer for a while. This is because the restricted mutation has some extra operations to identify and preserve rare branches, which may cost extra time. However, many deep paths are hidden in the regions where can only be exercised through rare branches. Although exploring these regions costs much time and lowers the speed, our mutation strategy has the ability to continue raising the metrics when AFL is hard to find new

paths and branches. Consequently, DeepFuzzer also triggers more crashes.

From these statistics and analysis above, we find that the hybrid mutation strategy is positive to fuzzing. It improves the coverage and facilitates hunting more vulnerabilities.

5.1.5 DeepFuzzer performance

We integrate the qualified seed generation, balanced seed selection, and hybrid seed mutation to form DeepFuzzer. Because they are complementary and serially connected, results get better when qualified seed generation, balanced seed selection and hybrid seed mutation are combined. Seed generation produces qualified seeds that reach deep regions of the program state space. Hybrid mutation strategy further explores the program on the basis of the reachability brought by initial seeds. Balanced selection assigns each seed a reasonable mutation time.

TABLE 4
Number of paths (two cores)

Project	AFL	AFLFast	FairFuzz	QSYM	MOPT	DeepFuzzer
boringssl	664	697	756	716	565	766
c-ares	29	30	29	34	31	36
guetzli	511	390	912	663	749	1638
lcms	277	318	266	293	234	296
libarchive	2238	2117	1573	1079	2032	2738
libssh	17	17	17	18	15	20
libxml2	2710	2358	2318	1870	5289	5827
pcre2	11038	10719	12110	11479	11332	12568
proj4	64	65	61	76	58	245
re2	2113	2087	2008	2413	1367	2213
ffmpeg	7676	7235	7394	1459	8110	9081
openjpeg	440	534	445	276	785	645
tcpdump	1670	3899	2498	500	5232	5265
Total	29447	30466	30387	20876	35799	41338

Table 4, 5, 6 compares the performance of DeepFuzzer, QSYM as well as other AFL-family fuzzers. Because QSYM requires an AFL instance to run simultaneously and needs at least two cores, we also run DeepFuzzer and other fuzzers with two cores. Besides Google fuzzer-test-suite, we also evaluate these fuzzers on three large real-world programs namely ffmpeg, openjpeg and tcpdump. The results show that DeepFuzzer outperforms other fuzzers including QSYM. Table 4 and Table 5 show that DeepFuzzer has a higher coverage than other fuzzers. Compared to AFL, AFLFast, FairFuzz, QSYM, and MOPT, DeepFuzzer executes 40%, 36%, 36%, 98%, and 15% more paths, covers 37%, 34%, 34%, 101%, and 11% branches respectively. From Table 6, we observe that DeepFuzzer triggers more unique crashes than other fuzzers. DeepFuzzer crashes 10 programs while AFL, AFLFast, FairFuzz, QSYM, and MOPT crash 5, 5, 5, 7, and 3 programs. According to the number of unique crashes triggered, DeepFuzzer achieves an improvement of 30%, 240%, 102%, 147%, and 257% compared to AFL, AFLFast, FairFuzz, QSYM, and MOPT. We also develop a component that analyzes these crashes by discriminating frames extracted from core dump files to get the number of bugs in Table 7, which have also been manually checked. It shows that DeepFuzzer finds 11 bugs while AFL, AFLFast, FairFuzz, QSYM, and MOPT find 5, 5, 5, 7, 5 bugs, respectively.

TABLE 5
Number of branches (two cores)

Project	AFL	AFLFast	FairFuzz	QSYM	MOPT	DeepFuzzer
boringssl	9680	9897	10733	9228	9228	11320
c-ares	277	268	277	277	277	277
guetzli	5544	3403	8420	5155	8568	32521
lcms	7327	7319	6724	6911	6211	7720
libarchive	24673	25318	18919	10635	23202	27191
libssh	1595	1592	1595	1595	1595	1595
libxml2	34281	33413	32546	26212	54453	57048
pcre2	75997	73711	76233	71208	73013	78552
proj4	594	592	592	594	588	2574
re2	32805	32538	32419	32665	29454	33025
ffmpeg	104618	98244	105500	46055	140297	139346
openjpeg	4071	4471	4062	2497	4912	5018
tcpdump	21479	38596	31377	5980	46050	45025
Total	322941	329362	329397	219012	397848	441212

QSYM outperforms other fuzzers such as Driller and VUzzer[20] in small scale projects like LAVA-M in its paper. And QSYM also applies many optimizations to scale symbolic execution to complex programs. However, in our experiments, QSYM fails to outperform DeepFuzzer in large real-world programs of Google fuzzer-test-suite. In particular, in three large programs namely ffmpeg, openjpeg, and tcpdump, DeepFuzzer covers 450% more paths and 191% more branches than QSYM. From Table 6, we observe that DeepFuzzer triggers more crashes than QSYM in 9 projects. Table 7 shows that DeepFuzzer finds three more bugs than QSYM. The main reason behind might relate to the size of the program. Most of the programs we tested have a large size than programs tested in QSYMs paper such as base64.

TABLE 6
Number of crashes (two cores)

Project	AFL	AFLFast	FairFuzz	QSYM	MOPT	DeepFuzzer
boringssl	0	0	0	0	0	0
c-ares	6	7	9	3	9	7
guetzli	0	0	0	0	0	2
lcms	20	13	48	29	0	7
libarchive	0	0	0	0	0	31
libssh	0	0	0	0	0	0
libxml2	22	7	8	3	65	19
pcre2	1361	510	840	699	439	1734
proj4	0	0	0	0	0	0
re2	3	1	1	0	0	6
ffmpeg	0	0	0	2	0	7
openjpeg	0	0	0	5	0	20
tcpdump	0	0	0	2	0	3
Total	1412	538	906	743	513	1836

It is well known that symbolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with the program size. To mitigate this problem, hybrid fuzzing offloads the trivial part of the exploration to fuzzing, and only utilizes the symbolic execution to solve the difficult part. Thus the symbolic execution engine can discard a large number of trivial paths and alleviate the path explosion. However, as the exponential path growth is intrinsic to symbolic execution, even as the optimizations in QSYM cannot either lower the asymptotic computational complexity, not to mention the stage of SMT solving, an NP-complete problem. When the program gets larger, the cost of tracing greybox fuzzing,

TABLE 7
Number of bugs (two cores)

Project	AFL	AFLFast	FairFuzz	QSYM	MOPT	DeepFuzzer
boringssl	0	0	0	0	0	0
c-ares	1	1	1	1	2	1
guetzli	0	0	0	0	0	1
lcms	1	1	1	1	0	1
libarchive	0	0	0	0	0	1
libssh	0	0	0	0	0	0
libxml2	1	1	1	1	2	1
pcre2	1	1	1	1	1	1
proj4	0	0	0	0	0	0
re2	1	1	1	0	0	1
ffmpeg	0	0	0	1	0	2
openjpeg	0	0	0	1	0	1
tcpdump	0	0	0	1	0	1
Total	5	5	5	7	5	11

collecting constraints, and solving constraints also increases. As a result, the performance gains brought by symbolic execution tend to diminish when employing QSYM in testing large programs. In contrast, DeepFuzzer adopts a more lightweight approach when invoking symbolic execution. By moving the symbolic execution forward to generate initial seeds, it controls the time and resource usage and eliminates the cost of switching between two techniques. DeepFuzzer also utilizes the balanced seed selection to fuzzing valuable seeds, and it will quickly cover a lot of "difficult" areas by hybrid seed mutation based on initial seeds. The results and the analysis demonstrate that when testing real-world software, utilizing symbolic execution to generate high-quality initial input seeds and fully utilize them in fuzzing might be a better combination of two techniques.

From these statistics and our findings explained in the analysis above, we find that DeepFuzzer has the ability to increase not only coverage, but also the probability of identifying vulnerabilities by triggering more crashes.

5.2 Industrial Practice

In order to evaluate DeepFuzzer in a real industrial environment, we collaborated with engineers from Huawei and deployed DeepFuzzer to fuzz `libmsg` developed there. `libmsg` is a proprietary message middleware which is responsible for transferring the message of the entire distributed system department. We encountered several obstacles that have failed some academic fuzzers, Driller, for example, takes two engineers two weeks of effort to configure but then failed. Benefiting from the toolchain of DeepFuzzer to support automatic fuzzing environment configuration, we overcame the obstacles and ran the program successfully. DeepFuzzer found 9 previously-unknown vulnerabilities, including flaws that lead to slow resource leak, denial of service, and immediate system crashes.

5.2.1 Experiment setup

The complete fuzzing procedure for DeepFuzzer consists of the preparing stage and the fuzzing stage. Steps of the preparing stage contain writing the fuzzing driver, preparing the environment, compiling hardened binary, and validating the fuzzing driver. When the preparation is down, DeepFuzzer starts to feed the program with a lot of seeds and monitor exceptions in the fuzzing stage. In order to boost the convenience of the fuzzing process, DeepFuzzer tries to automate all the steps as much as possible with the toolchain module. The only thing that the test engineers need to do manually is writing the corresponding fuzzing driver, which exposes an interface to DeepFuzzer for feeding the input and execution. The external network interface is chosen to focus on testing the protocol handlers of `libmsg`. Choosing this interface brings us three benefits: completeness for the reachability of almost all code paths of `libmsg`, convenience for feeding input, and accuracy for identifying vulnerabilities.

5.2.2 Typical obstacles and solutions

After choosing the interface, things did not go as smoothly as we planned. We almost stumbled at each step due to the complexity and diversity of the industrial environment. Typical obstacles that would fail other fuzzers are listed as follows:

The first obstacle is the **system build complexity**. `libmsg` has a complex build procedure. Due to the historical burden, the build scripts are a mix of bash, autoconf, and CMake. This situation is also common for complex projects which have many dependencies in companies. It is tedious to manually modify a series of scripts to enable sanitizers. We solved it with the toolchain component of DeepFuzzer. The toolchain component can recognize different build systems and inject appropriate instrumentation-related compiler options.

The second obstacle is the **shallow bugs**. When validating the fuzzing driver, the program crashed on virtually every seed, even the most simple ones constructed by hand. The shallow bugs hindered further operations. We solved it by collaborating with the developer to fix these bugs.

The last obstacle is the **bug-hiding code segments**. As a messaging library, `libmsg` is expected to handle errors and run continuously until explicitly told to exit. It simply disables the error reporting and directly exits with status code 0. Many projects also have a similar mechanism for maintaining continuous availability. But the mechanism can also mislead fuzzers because they rely on an exit code to identify crashes. The solution is to remove the logic which disables the error reporting mechanism and patch DeepFuzzer to detect a non-zero exit code.

5.2.3 Results

After overcoming those obstacles, we successfully ran DeepFuzzer on `libmsg`. Benefiting from the adapting experience and the toolchain component of DeepFuzzer, we also successfully adapted AFL in the same setting.

Table 8 shows the existing known vulnerabilities and the results of AFL and DeepFuzzer. It illustrates that DeepFuzzer performs better. AFL and DeepFuzzer detects 5 and 11 vulnerabilities, respectively. Two known vulnerabilities are found by both fuzzers. In other words, AFL and DeepFuzzer reveal 3 and 9 unknown vulnerabilities, respectively.

For the only one memory leak vulnerability that is not captured by both AFL and DeepFuzzer, we find that they may be unable to cover some regions of the program, limited by the simple fuzzing driver which converted from the

TABLE 8
Number of vulnerabilities Detected

Type	Known	AFL	DeepFuzzer
Incorrect exit procedure	0	2	2
File descriptor leak	0	1	1
Reachable assertion	0	0	1
Unaligned memory access	2	2	2
Uncontrolled memory allocation	0	0	4
Use after free	0	0	1
Memory leak	1	0	0
Total	3	5	11

simple sample code. Manual tests may have the advantage of having knowledge of the program, and they can systematically cover most of the interfaces and APIs. We believe a better driver will help DeepFuzzer solve this problem and capture more vulnerabilities. Nine other previously unknown vulnerabilities are captured by DeepFuzzer. For example, the file descriptor leak can only be triggered when all file descriptors are exhausted by feeding a large number of inputs. However, manual tests might never detect this problem because it usually feeds the program with a small number of predefined inputs.

From these statistics and above analysis, we find that DeepFuzzer can be applied in industrial practice to hunt real bugs. With the experience gained from overcoming those obstacles, it is easier to apply fuzzing on other real-world projects. These experiences and the developed toolchain can not only help DeepFuzzer, but also have some reference value for adapting other fuzzers on real projects.

5.3 Discussion

There are some reasons for limiting the use of DeepFuzzer in practice. We provide below some potential methods to enhance its functioning.

Unlike static analyzers working on source code, fuzzing is a dynamic method. Fuzzers that cannot be executed in the environment of the target program are worthless. The complex environment and configurations, including the operating system, the compiler, and the hardware, are usually vastly different thus may lead to configuration inconsistency. DeepFuzzer and many advanced academic technologies are usually limited by environmental inconsistency and diversity, resulting in unavailability. In order to improve the utility, DeepFuzzer has a support layer in the toolchain, so it can run in almost all recent Linux versions. Despite this, it might still be unavailable because of configuration failed. For example, the seed generator of DeepFuzzer does not support multi-threading and inline assembly code, and we have to disable seed generation for programs that have these features.

Missing the fuzzing driver is another threat for efficient fuzzing. Developing a fuzzing driver requires sufficient domain knowledge as well as adequate understanding of fuzzing technology. Unfortunately, typically people who develop fuzzers lack domain knowledge while developers with domain knowledge do not understand fuzzing. To solve this, promoting fuzzing technology is necessary, the developer needs to put writing the fuzzing drivers as a step in the development process. On the other hand, to lower the cost, converting sample code or unit tests to form fuzzing drivers might be a convenient method.

In addition, some of the programs being tested use low-level interventions such as the signal handler, which prevents DeepFuzzer and other fuzzers from detecting anomalies. These workarounds are hard to locate in the source code but must be found and bypassed. Besides detecting crashes, fuzzers can also monitor the usage of resources. A resource utilization anomaly may also imply a bug. For example, after investigating the exceedingly high number of execution timeouts, we find the uncontrolled memory allocation vulnerability. When the allocation is too large, the kernel will over-commit memory, which slows down the memory accesses. There are also some limitations in other aspects, for example, DeepFuzzer cannot divide unique crashes caused by the same bug in real-time. Its crash analyzer component can only work after the fuzzing process.

6 Related Work

For the related work, we focus on generation-based fuzzing, mutation-based fuzzing, and symbolic execution-based fuzzing. We also discuss the differences between these works and our study.

6.1 Generation-Based Fuzzing

Generation-based fuzzing generates inputs from scratch based on specifications. For programs which require inputs in complex formats, this fuzzing technique is especially useful. The specification can be separated into two categories including input model and context-free grammar.

Model-based fuzzers [21], [22], [9] employ a model of the protocol. The model is executed on- or off-line to generate complex interactions with the test program. Peach [22] is an input model-based fuzzer and combines generation and mutation capabilities. Peach operates by applying fuzzing to models of the data and the state. The data-model describes the format of the input, and the state-model defines the ways of interacting the data with the fuzzing targets. Skyfire [9] generates inputs from a context-sensitive grammar model, which is learned from corpora and grammars.

Many fuzzers [10], [23], [24], [25] generate inputs based on context-free grammar. Csmith [10] is a C-compiler test tool which can generate random C programs conforming to the C99 standard. This tool has the ability to generate programs exploring typical combinations of C-language features while free of undefined and unspecified behaviours. Sirer and Bershad developed lava [23] to generate effective test suites for JVM. IFuzzer [24] takes the context-free grammar of a language as the input to generate parse trees and to extract code fragments from a given test-suite. Then it generates new code fragments by performing genetic operations on the parse tree.

Some tools try to automatically learn the specification [26], [27], [28]. Glade [26] automatically synthesizes a context-free grammar from seeds and blackbox program access, and then uses the synthesized grammar in conjunction with a standard grammar-based fuzzer to generate new test inputs. Learn&Fuzz [27] employs neural-network-based

statistical learning to automatically generate input grammar from sample inputs.

DeepFuzzer is a mutation-based fuzzer, so it differs greatly from these tools. Though DeepFuzzer also generates inputs with lightweight symbolic execution, these inputs play the role of the initial seeds to supply start points for mutational fuzzing, and are further mutated and scheduled based on coverage information.

6.2 Mutation-Based Fuzzing

Mutation-based fuzzing is a basic way to detect vulnerabilities when the fuzzer knows little about the program. It generates input seed by modifying the existing inputs.

Many tools [8], [12], [14], [29], [20], [13], [30] apply various techniques to boost the fuzzing process. AFLFast, AFLGo, CollAFL, and VUzzer focus on improving the seed selection. AFLFast [12] proves that the process of AFL can be regarded as a Markov chain. A power schedule computes the times of random mutation for each seed. AFLFast carries out a fast power schedule based on the path frequency of the seeds executed. Similar to AFLFast, AFLGo [14] implements a simulated annealing-based power schedule, which helps to fuzz some target areas in the codes. In particular, it proposes a way to measure the distance between the seeds and the targets. CollAFL [29] provides more accurate coverage information to mitigate path collision. It also utilizes the coverage information to apply some seed selecting strategies. VUzzer [20] utilizes static analyze to obtain the control structure and prioritizes seeds which execute deep paths. It also uses taint analysis to help mutate seeds. FairFuzz [13], ProFuzzer [30], and MOPT [15] focus on improving the seed mutation. FairFuzz only mutates seeds which hit rare branches and it strives to ensure the mutant seeds still hit the rarest ones. ProFuzzer probes the input fields (for example, assertion, raw data, off-size) and adapts the mutation strategy to them. MOPT schedules the mutation operators based on Particle Swarm Optimization.

Compared with these tools, DeepFuzzer considers the seed generation, seed selection, and seed mutation make up a whole fuzzing process and coordinates them to gain an edge over other fuzzers. The process of seed generation is lacked in all of these tools. For seed selection, DeepFuzzer controls the mutation times like AFLFast and AFLGo. The difference is that DeepFuzzer attributes the times by branch frequency, which associates with the state space of the program more tightly than path frequency. CollAFL, VUzzer and DeepFuzzer all collect the branch coverage, but input selection schemes are different. CollAFL utilizes the topology of blocks to calculate the number of untouched neighbour branches when fuzzing, which determines the weight of an input. Similarly, VUzzer also utilizes the topology to assign higher weights to deeply nested basic blocks. Then it prioritizes deep paths by selecting seeds based on the sum of the weights of all the basic blocks covered. Differently, DeepFuzzer is unaware of the structure of the target program. It utilizes the branch count statistics to infer rare branches and calculate the weights.

For seed mutation, DeepFuzzer, FairFuzz, and ProFuzzer all recognize the critical bytes which influence the execution behaviour of the target problem through per-byte mutation. But they differ in details. DeepFuzzer applies the hybrid mutation strategy which combines the random and the restricted mutation strategies. Compared with FairFuzz which only applies reserved mutation, the random mutation of DeepFuzzer preserves the probability of exploring other regions so that the local convergence of FairFuzz is unlikely to occur in DeepFuzzer. Compared with ProFuzzer, the mutation of DeepFuzzer is lightweight. ProFuzzer replaces each byte with all 256 values to extract features and infer the fields. In contrast, DeepFuzzer only replaces each byte with one random value to infer the delicate structures of the input. As a result, ProFuzzer has an additional stage to probe the input fields while DeepFuzzer integrates it into the mutation stage. MOPT focuses on how to select mutation operators. Different from it, DeepFuzzer concentrates on refining mutation operators to improve mutation quality.

6.3 Symbolic Execution-Based Fuzzing

Symbolic execution treats the input of the program as symbols, collects the constraints to generate symbolic formulas and employs constraint solvers to find out concrete inputs. Godefroid presents an introduction to automatic input generation using symbolic execution [31]. Some tools apply pure symbolic execution [32], [33], [34] while others integrate it with other techniques [17], [35], [36].

SAGE [32], KLEE [33], and Mayhem [34] are pure symbolic execution tools. SAGE uses the generational search that better leverages expensive symbolic execution. It collects constraints by actually running the program on well-formed inputs, then negates the constraints one by one and solves them to produce inputs. KLEE is a refined version of EXE [37]. It employs a variety of constraint solving optimizations and uses a heuristics search to get high code coverage. KLEE runs on-line — it forks on branching in a single run, which differs from SAGE — working on execution traces off-line. Mayhem combines them by alternating between on-line and off-line. It can automatically find exploitable bugs in binary programs. Although these tools are powerful, the path explosion problem is still inevitable.

Driller [17] integrates symbolic execution and mutation-based fuzzing to mitigate the path explosion problem. It utilizes quick fuzzing to explore most of the paths and switches to symbolic execution to solve complex constraints when the fuzzer gets stuck. QSYM [18] makes hybrid fuzzing scalable enough to test real-world applications. VeriFuzz [38] combines fuzzing with static analysis, and it uses constraint-solver to generate initial seeds. Some tools [35], [36] combine taint analysis and symbolic execution. The basic idea is guiding symbolic execution towards vulnerable parts found by taint analysis. They work on a limited class of bugs and need the user to specify attack points.

Compared with pure symbolic execution tools, DeepFuzzer strictly limits the constraint solving time to prevent wasting too much computing resources with unnecessary explorations. Beyond this, DeepFuzzer utilizes symbolic execution only in the beginning to produce initial seeds. The essential difference between DeepFuzzer and Driller or QSYM is the usage of symbolic execution. DeepFuzzer utilizes it for generating high-quality seeds while Driller or QSYM applies it to overcome complex checks when fuzzing

gets stuck. VeriFuzz also generates seeds with constraint-solver. But it and Driller use a blind mutation strategy. Although they pass checks with the help of symbolic execution, they are likely to get stuck again. In contrast, DeepFuzzer has the ability to reserve the result of the symbolic execution by only mutating bytes at limited positions in restricted ways.

In the evaluation section, we mainly compared DeepFuzzer with AFL, AFLFast, FairFuzz, QSYM, and MOPT. They are chosen because they are the most relevant and are state-of-the-art fuzzers. For example, QSYM is the most recently developed symbolic execution aided fuzzing tool, and performs better than AFL and Driller as stated in its paper [18]. For a fair comparison, some other recent works are excluded. They include some works which are not much related, such as Skyfire, a generation-based fuzzer, or fuzzers that have different purposes, such as AFLGo, a patch testing fuzzer, or that are known not to be high performing, such as FidgetyAFL, or that have engineering problems, as with Driller. It took us one month to try to customize Driller on Google benchmark, but we failed. Driller is designed for testing the CGC binaries in cluster. The property of cluster fuzzing implies the high cost of configuring the environment, for example messaging middlewares and databases. The property of CGC-oriented implies the lack of environment modelling and system call support. It is impossible to run applications in Google fuzzer-test-suite when emulation of mmap() is missing. In addition, Driller only supports standard input, but these applications all require file input. LibFuzzer is also excluded because of the engineering problem. libFuzzer will shut down when it finds a crash, while other tools continue fuzzing unless manually terminated.

7 Conclusion

In this paper, we propose DeepFuzzer, which applies symbolic seed generation, balanced seed selection, and hybrid seed mutation to the acceleration of deep greybox fuzzing. The light-weight symbolic execution supplies high-quality initial seeds which contain abundant mutation information. The seed selection and mutation take full account of branch coverage balance to decide how to select and mutate seeds to ensure depth and breadth. Evaluated on Google fuzzer-test-suite, DeepFuzzer outperforms other commonly used fuzzers, and it is capable of exploring a wider program state space and triggering more crashes. In addition, we apply DeepFuzzer in a complex industrial development environment to fuzz a real-world program `libmsg` of Huawei. We set out the main obstacles and relevant solutions. After overcoming these obstacles, DeepFuzzer performed well and reported 9 previously-unknown bugs. This engineering practice and the developed toolchain provide some references for fuzzing other industrial software. Our future work focuses in two directions. The first is to support concurrency for the seed generator, and the other is to integrate the resource utilization to guide greybox fuzzing.

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1995.

[3] "libFuzzer in Chrome," https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/README.md, 2017, [Online; accessed 12-November-2017].

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

[5] "SDL PROCESS: VERIFICATION," https://www.microsoft.com/en-us/SDL/process/verification.aspx, [Online; accessed 26-January-2018].

[6] "Microsoft Security Risk Detection ("Project Springfield")," https://www.microsoft.com/en-us/research/project/project-springfield/, 2015, [Online; accessed 26-January-2018].

[7] "A Basic Distributed Fuzzing Framework for FOE," http://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html, 2012, [Online; accessed 28-January-2018].

[8] M. Zalewski, "American fuzzy lop," 2015.

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

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[12] 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.

[13] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 475–485.

[14] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)*, 2017.

[15] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1949–1966.

[16] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 61–64.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[18] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[21] "Fuzzer Automation with SPIKE," http://resources.infosecinstitute.com/fuzzer-automation-with-spike/, [Online; accessed 12-February-2018].

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

[23] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *ACM SIGPLAN Notices*, vol. 35, no. 1. ACM, 1999, pp. 1–13.

[24] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 581–601.

[25] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments." in *USENIX Security Symposium*, 2012, pp. 445–458.

[26] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 95–110.

[27] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[28] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 720–725.

[29] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[30] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery," in *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE, 2019, p. 0.

[31] P. Godefroid, "Test generation using symbolic execution," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[32] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[33] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[34] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 380–394.

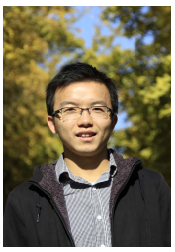
[35] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations." in *USENIX Security Symposium*, 2013, pp. 49–64.

[36] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 87–97.

[37] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

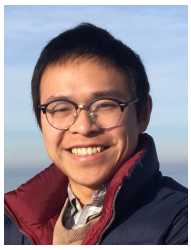
[38] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "VeriFuzz: Program aware fuzzing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 244–249.

Jie Liang received the BS degree in computer science from Beijing University of Posts and Telecommunications, Beijing, China, in 2017. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests include program analysis and its applications to industry.

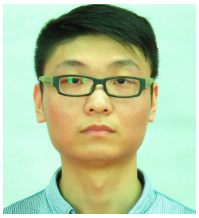


Yu Jiang received the BS degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He worked as a Postdoc researcher in the department of computer science of University of Illinois at Urbana-Champaign, IL, USA, in 2016, and is now an assistant professor in Tsinghua University. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

Mingzhe Wang received the BS degree in software engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2018. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests include program analysis and its applications to industry.



Xun Jiao is an assistant professor in the ECE department of Villanova University. He obtained the Ph.D. degree from the department of Computer Science and Engineering at the University of California, San Diego. He received the dual bachelors degree from the Beijing University of Posts and Telecommunications, China and the Queen Mary University of London, United Kingdom, in 2013. His research interests include error-tolerant computing and machine learning.



Yuanliang Chen received the BS degree in software engineering from Nanjing University, Nanjing, China, in 2017. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests include program analysis and its applications to industry.

Houbing Song received the PhD degree in electrical engineering from the University of Virginia, Charlottesville, VA, in August 2012. In August 2017, he joined the Department of Electrical, Computer, Software, and Systems Engineering, Embry-Riddle Aeronautical University, Daytona Beach, FL, where he is currently an Assistant Professor and the Director of the Security and Optimization for Networked Globe Laboratory (SONG Lab, www.SONGLab.us). His research interests lie in the areas of cyber-physical systems, the Internet of things, cloud computing, big data, connected vehicles, wireless communications and networking, and optical communications and networking.

Kim-Kwang Raymond Choo received the Ph.D. in information security in 2006 from Queensland University of Technology, Australia. He is currently a cloud technology endowed associate professor at University of Texas at San Antonio, an associate professor at the University of South Australia, and a guest professor at China University of Geosciences. He is the recipient of various awards including ESORICS 2015 Best Research Paper Award, Winning Team of Germanys University of Erlangen-Nuremberg Digital Forensics Research Challenge 2015, and 2014 Highly Commended Award by the Australia New Zealand Policing Advisory Agency. His research interests include cyber security and forensics.