Semantic Sensitive Coverage-based Fuzzing

Jun LI and Chao ZHANG*
Tsinghua University, China
*Corresponding author

Keywords: Fuzzing, Coverage-based, Semantic sensitive, Software security.

Abstract. Coverage-based fuzzing is widely used in finding program bugs. While state-of-the-art coverage-based fuzzers, either ignore the differences between newly discovered edges or consider only control flow features (e.g., depth) when prioritizing seeds for mutation. In this paper, we propose a semantic sensitive coverage-based fuzzing solutions, SSFuzzer. When new edges are discovered during fuzzing, it evaluates the semantic features of the new edges and update the weights of testcases. Seeds with heavier weights will first be picked to mutate and be given more energy to mutate (i.e., more testcases will be generated). We evaluate not only positive semantic features (e.g., memory access) but also negative ones (e.g., error handling) of edges. We implemented a prototype based on AFL. Experiment results demonstrate that SSFuzzer can discover vulnerabilities faster.

Introduction

Fuzzing is a popular method in finding software bugs. To achieve better results, a variety of fuzzing strategies are used. Previous work has proved that better code coverage results in higher probability of finding bugs [5]. Thus, state-of-the-art fuzzers like AFL [7] and Vuzzer [4] employ a coverage-based fuzzing strategy. More specifically, program control flow is interpreted as a graph, and vertices are used to represent basic blocks, edges are used to represent transitions between the basic blocks. Coverage-based fuzzers usually maintain a feedback loop, and retain inputs that discover new paths (edges) for further mutations.

We analyzed some state-of-the-art coverage-based fuzzers, and summarize the strategies they use to achieve higher coverage. AFL hashes the edges of a program and records the edge hit count in a shared memory. For any control flow edges it exercises, it prioritizes the smallest and fastest testcase for mutation. However, it treats all newly discovered edges equally and select seeds in sequence, thus cannot select the most promising ones to mutate. AFLFast [3] and Vuzzer [4] improve the seed selection strategy to improve the efficiency or effectiveness of bug finding. AFLFast measures the frequency of all exercised paths, selects seeds that have been fuzzed fewer, and allocates more energy to seeds that exercise low-frequent paths. Vuzzer integrates static and dynamic analysis to identify hard-to-reach deeper paths, and gives a higher priority to seeds exercising deeper paths. Both of these works focus on the control flow features of paths, ignoring the semantic features, e.g., what functionality the code implements. We observed that deeper paths may not always be the better ones, but code with dangerous semantic operations are more promising to cause problems. And thus, we should consider semantic features to fuzz.

In this paper, we propose a semantic sensitive coverage-based fuzzing solution SSFuzzer. It evaluates the semantic features of edges and updates the seeds’ weights accordingly. Semantic features are collected through static and dynamic analysis. Three main semantic features in the code are considered currently: dangerous function calls, memory access frequency and error handling operations. Edges are assigned with different scores based on these features. Then we calculate the weights of newly discovered paths based on the edge scores accordingly. Seeds with more weights are first picked to mutate and will be given more energy to mutate.

We implemented a prototype of SSFuzzer as an extension to AFL, and evaluated it on several widely used applications in Linux. Each application is tested for 24 hours with the same initial seed.
inputs. Though the testing time is very short for a usual fuzzing test, 384 unique crashes are found in 5 applications. Results show that SSFuzzer is able to discover vulnerabilities faster than AFL.

In summary, we make the following contributions:

[1] We analyze the state-of-the-art fuzzers extensively, and summarize the shortcoming of previous works.

[2] We propose a semantic sensitive coverage-based fuzzing solution SSFuzzer, taking the semantic features into consideration when prioritizing seeds for mutation.

[3] We implement a prototype of SSFuzzer. Experiments demonstrate that our solution is able to discover vulnerabilities faster.

Design

![Figure 1. Architecture of SSFuzzer.](image)

Figure 1 depicts the architecture of SSFuzzer, consisting of three main phases: (i) Reverse Engineering, which decompiles the binaries instrumented by AFL; (ii) Feature extracting, which extracts semantic features of binaries with static and dynamic analysis, and generate a mapping for edges and weights; (iii) Fuzzing, the main fuzzing loop, which selects seeds with more weights to mutate and gives more energy for mutation and testing.

Semantic Features

Control flow and data flow features have been used to guide the fuzzing process \([4]\), while semantic features are rarely considered. We observe that some program operations are more likely to fail the program or introduce a vulnerability, while other operations are not. We believe that these features can be used to guide the fuzzing process to find vulnerabilities faster.

Types of Semantic Features. We consider only three features in our current prototype. We believe that more features could be found and utilized to guide fuzzing.

Dangerous function calls. Previous work showed that dangerous function call is a one of the root causes of software vulnerabilities. Improper uses of these functions may result in high-risk program vulnerabilities, such as buffer overflow and information disclosure.

<table>
<thead>
<tr>
<th>Dangerous function</th>
<th>Number of CVEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcpy</td>
<td>33</td>
</tr>
<tr>
<td>sprint</td>
<td>21</td>
</tr>
<tr>
<td>strcpy</td>
<td>15</td>
</tr>
<tr>
<td>sscanf</td>
<td>11</td>
</tr>
<tr>
<td>memset</td>
<td>9</td>
</tr>
<tr>
<td>strcat</td>
<td>7</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>7</td>
</tr>
<tr>
<td>realpath</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1 shows the number of CVE \([2]\) vulnerabilities caused by dangerous C function calls since 2000. Basic blocks containing dangerous function calls are more likely to be vulnerable. As to cases
that causing of bugs are not in the dangerous function calls basic blocks while just crashed on the function call point, testcases reaching these function calls are still more likely to reach a bug state than those do not reach such basic blocks. And thus, we assign more weights to this type of blocks and edges terminating with such blocks. It is likely that vulnerabilities could be found faster if more energy is given to seeds that exercise such edges.

**Memory access frequency.** Memory safety violations happen at memory access operations. We believe that seeds with higher frequency of memory access operations are more promising to trigger crashes. For instance, most memory corruption vulnerabilities like stack overflow and heap overflow, are mainly caused by improper memory access. Allocating more energy on testcases with higher frequent memory access option can improve the probability of discovering such vulnerabilities. Thus we give more weights to edges that terminate with basic blocks containing more memory access operations.

**Error handling operations.** Blackbox mutation-based fuzzing inevitably results in large amounts of invalid inputs, which are often discarded or rejected by error handling code of target applications. Error handling code is thus frequently hit during fuzzing, while it contributes very little in finding new bugs. Recognizing and deprioritizing such code can reduce time wasting and accelerate the generation of interested inputs. We thus give edges terminating with error handling code a relatively smaller weight value.

**Feature Extraction.** The aforementioned semantic features are extracted by static and dynamic analysis. Before the fuzzing process, we perform reverse engineering and static analysis on target binary applications instrumented by AFL. We identify dangerous function calls and memory access instructions by scanning the binary code, and assign basic blocks with these dangerous operations more weights.

It is worth noting that, AFL already instruments a code snippet in each basic block to compute the hash of the incoming edge, in order to keep track of the runtime code coverage. We utilized this mechanism used by AFL, to also keep track of the weight of the incoming edges, i.e., the weight of the current basic block.

Dynamic analysis is used to collect features relate to runtime program behaviors. And in this paper we only considered the error handling code. We recognize such code in a similar way as Vuzzer. First, we collect a set of valid inputs $S_v$ and randomly generate a set of inputs $S_r$ of which most are invalid. Then, we feed both sets of inputs to the target program and record the edges they hit $E_v$ and $E_r$ separately. An edge is considered to be related to error handling code if it exists in the paths of most random inputs while not exist in any valid ones.

**Seed Selection Strategy**

A coverage-based fuzzer usually collects runtime code coverage information through lightweight instrumentation on binaries. For example, AFL instruments basic blocks to keep track of edge transitions. If a testcase hits a new edge or change the hit count of an edge significantly, AFL queues it in a seed pool for further mutation and fuzzing. Smaller and faster seeds from the pool are given more energy to mutate, while edges are treated equally.

---

**Algorithm 1.** SSFuzzer Seed Selection Strategy.

```
Input: Seed Input $S$, Weight $W$
1: $T = S$
2: $T_x = \emptyset$
3: repeat
4:   $t = \text{choose\_next}(T)$
5:   $s = \text{calculate\_weight}(t)$ # energy
6: for $i$ from 1 to $s$ do
7:   $t' = \text{mutate}(t)$
8:   if $t'$ crashes then
9:     $T_x.add(t')$
10: else if isInteresting($t'$) then
11:     $t'.\text{weight\_change} = \text{mapping\_weight}(W,t')$
```

---
Instead of treating all these testcases indiscriminately, SSFuzzer prioritizes seeds that exercise paths containing new edges with more weights. If a testcase is of interest, i.e., new path is exercised, SSFuzzer assign the testcase a weight according to all newly discovered edges on its exercised path. More Weights represent greater potential for discovering vulnerabilities, and result in a higher score when computing the energy of mutation.

Algorithm 1 shows the general work flow of our seed selection strategy. It computes the energy of the seed inputs depending on their exercised edges, and adjust the mutation air time of each seed accordingly.

### Experiment and Evaluation

To evaluate the effectiveness of our seed selection strategy, a set of experiments are conducted. We first evaluated the semantic feature extraction and show the prevalence of edges consists of sensitive semantic features. Then we experiment the seed selection strategy on a series of real world softwares and show the crash finding ability.

#### Feature Extraction

To evaluate the prevalence of the semantic features in applications, we take the dangerous function call as an example. We performed our analysis on the latest version of Coreutils \(^{[1]}\) by the time of writing. Coreutils is a set of commonly used command line utilities of the GNU operating system. We analyzed all 105 utilities and present the analysis result of randomly selected 30 programs.

Figure 2 presents the percentage of edges with dangerous C function calls in selected programs. It shows that at least 1% of edges are dangerous. More specifically, when we hit a new edge during a coverage based fuzzing process, the probability of hitting an edge with dangerous function calls is about one percent. Thus allocating more energy on these edges could improve the probability of reaching such edges and reduce waste of computing resources. This percentage value can also be used to determine the weight value of such edges during fuzzing.
Crash Detecting

We performed our testing on several widely used real world applications on Linux like pngquant and libtiff. Each test was run on a X86-64 Linux server for 24 hours both with original AFL and SSFuzzer. We evaluate the ability of detecting crashes of semantic sensitive fuzzing by comparing the detecting results. In total, we found 384 unique crashes.

Figure 3 presents the crash numbers of tested applications. In the testing, SSFuzzer found 32 crashes in pngquant and AFL found 13 crashes. SSFuzzer found 82 crashes in tiffsplit and AFL found 57 crashes. For tiff2bw, SSFuzzer found 2 crashes and AFL didn’t find any. The result shows that our seed selection strategy is able to discover more crashes in the same testing time, faster.

We also calculated the average number of executed testcases to discover each crash of these softwares. The result is presented in Figure 4. For tiff2bw, as original AFL didn't discover any crash, we use a relatively bigger value instead, for a better visualization. SSFuzzer needs to perform weight value query operations frequently during the fuzzing loop, which inevitably results in a slowdown in execution speed. Despite this, SSFuzzer found more crashes than AFL in the same testing time. The results show that SSFuzzer uses less testcases to discover more crashes, and demonstrates that semantic sensitive features can guide the fuzzing process and improve the efficiency.

What's more, we calculated the average number of executed paths and present the result in Figure 5. Still, we use a relatively larger value for tiff2bw here. The result is quite similar to the previous one. It's reasonable for a coverage-based fuzzer, for which the number of testcases and executed paths are positively associated. The results show that semantic sensitive fuzzing solution makes coverage-based fuzzing more efficient and increased the probability of discovering bugs on the executed paths.

Conclusions

In this paper, we proposed a semantic sensitive coverage-based fuzzing solution and implemented a prototype, SSFuzzer. Different from previous coverage-based fuzzers, SSFuzzer collects semantic features of edges in programs, and allocate energy to seeds accordingly. It thus has a higher probability in discovering vulnerabilities. We proposed three semantic features and used them in SSFuzzer. Experiments demonstrate that SSFuzzer can detect vulnerabilities faster than existing fuzzer AFL, especially in detecting semantic features related vulnerabilities.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under the grant No. 61772308, the CCF-NSFOCUS Kunpeng Award, and the Young Talent Development Program by CCF.

References


