Vulnerability Detection in SIoT Applications: A Fuzzing Method on their Binaries

Xiaogang Zhu, Sheng Wen, Alireza Jolfaei, Mohammad Sayad Haghighi, Seyit Camtepe, and Yang Xiang

Abstract—SIoT enables devices to communicate with each other automatically, which is not reliable when SIoT applications are vulnerable themselves. To improve the security of SIoT, different techniques have been employed so far, mainly to detect vulnerabilities in applications. Among the detection techniques, fuzzing is one of the most effective ones that can significantly improve the security of SIoT applications. However, the existing fuzzing methods have three problems. First of all, the schemes to instrument target binaries cause high memory overhead because they instrument at all edges to obtain the coverage information. Moreover, they introduce a severe problem called edge collision, i.e., two different edges are deemed the same during fuzzing. Thirdly, none of the existing fuzzers conduct fuzzing using path coverage because path coverage has high memory overhead. In this paper, we propose BECFuzz to resolve the above three problems. BECFuzz instruments at specific edges, and conducts fuzzing based on both edge coverage and path coverage, which greatly improves its effectiveness. We implement our BECFuzz based on two typical fuzzers which are widely recognised as baselines, AFL and AFLFast, and run experiments on 18 real-world programs. The results demonstrate that our method suppresses the state-of-art fuzzers in performance.

Index Terms—Social Internet of Things, IoT Applications, Security, Fuzzing, Edge Coverage, Path Coverage.

1 INTRODUCTION

Internet of Things (IoT) is becoming increasingly popular and important in daily activities of human beings. IoT objects can communicate and provide services to each other, including software services. They form social relationships not only in the owners’ layer, but also at the lower device-to-device service provisioning layer. Social Internet of Things (SIoT) integrates the concepts from social networking with IoT [1]–[3]. This integration enables IoT objects to interact with their desired service providers or other objects by themselves, without requiring humans to intervene [4]. Due to the nature of automatic and interdependent connections in SIoT, security of applications/software in the devices becomes critical and a matter of global reliability. Vulnerabilities in SIoT devices can result in the leakage of high-value information. Many applications in SIoT devices are not secure because more and more applications are developed based on open-source frameworks, which may be vulnerable. Meanwhile, the developers who do not have security knowledge may also introduce vulnerabilities into the applications. Moreover, some SIoT devices have limited computational resources, resulting in simple internal logic for applications. If any vulnerability exists in these applications, it can be detected with little effort. Therefore, effective detection methods are required to find the vulnerabilities.

Among all the vulnerability detection techniques, fuzzing is widely used due to its effectiveness and simplicity. As source code is typically not available for vulnerability analysis of commercial software and legacy applications in SIoT, in this research, we focus on detecting vulnerabilities in executable binaries by utilising fuzzing.

Fuzz testing (or fuzzing) is an effective technique to expose bugs or vulnerabilities. It generates a large number of test cases for target programs, and monitors the executions for exceptions such as crashes. The exceptions in target programs are potential bugs or vulnerabilities. To guide the generation of test cases, many efficient solutions have been developed. For instance, directed fuzzers [5], [6] intend to generate test cases that can reach the pre-determined target locations. Among all the fuzzing solutions, coverage-guided fuzzing [7]–[13] is one of the most effective solutions. Most coverage-guided fuzzers utilise the information of edges to perform fuzzing because edges reach certain balance between efficiency and coverage information [13]. Edge coverage-guided grey-box fuzzing (ECGF) differentiates edges by using edge identifiers, and inputs are saved as seeds based on these identifiers. The existing ECGFs first assign each block an identifier [8]–[12], and then calculate edge identifiers based on block identifiers. This strategy introduces a severe problem that two different edges can have the same identifier, which is so called edge collision. Recently, CollAFL [13] tried to mitigate the collision by searching specific block identifiers to assign unique edge identifiers based on accurate control flow graphs (CFGs). However, constructing accurate CFGs from binaries is a big challenge in the field of binary analysis. Therefore, in this paper, we propose a more effective method to fuzz applications, including the SIoT ones.

To record and differentiate edges while fuzzing, existing ECGFs regard edge identifiers as indices of a hash table (In
AFL [8], this hash table is called *bitmap*). The bytes in the bitmap indicate whether the associated edges are examined or not. Therefore, in order to avoid edge collision, the size of the bitmap should be larger than the number of edges. As analysed in CollAFL, the execution speed of fuzzing drops if the size of bitmap increases. If the number of edge identifiers is larger than the size of bitmap, edge collision will exist which in turn reduces the fuzzing performance. Existing fuzzers assign identifiers to all edges in target programs, wasting space in bitmap. Some edges always appear together with other edges, thus we do not have to assign identifiers to such edges.

Based on the bitmap, existing fuzzers calculate identifiers for the *coarse path* to delete useless data of the inputs (i.e., if two inputs exercise the same edge coverage, fuzzing chooses the shorter one as a seed) or to count the path frequency (e.g., AFLFast). Coarse path phrase was suggested because bitmap lacks information about the order of edges in a path. If fuzzing saves seeds based on coarse path, it will obtain many more seeds than when edge identifiers are used. The challenge is that the coarse-path-based solutions cause high time and memory overheads, as they will obtain too many seeds to effectively discover more code coverage. Coverage-guided fuzzers guide fuzzing to more code coverage based on seeds. The basic assumption is that inputs mutated from seeds can exercise more execution paths. However, too many seeds may have the problem that inputs mutated from different seeds exercise the same execution path, which reduces the efficiency of fuzzing. Fuzzing is a security tool that cares more about the crashes than the inputs. Therefore, in this paper, we will research on how to fuzz based on coarse path coverage.

Existing fuzzers instrument target binaries either statically [14], [15] or dynamically [16], [17] to obtain edge coverage information. The binaries utilising static instrumentation execute faster than those utilising dynamic instrumentation. When fuzzing, a higher execution speed will save time to test more inputs, which in turn increases the chance to have more coverage. Therefore, static instrumentation performs better than dynamic instrumentation in terms of execution speed. However, it is hard for a static analysis to get all edges accurately [18], [19], while dynamic analysis can accurately get all edges, including invisible edges in static analyses. Invisible edges are the ones that cannot be obtained during static analysis.

In this paper, we propose BECFuzz to efficiently and effectively perform fuzzing on binaries of SIoT applications. BECFuzz incorporates direct instrumentation on edges, which helps to resolve the problem of edge collision. Using the edge instrumentation, BECFuzz can differentiate edges in target programs. Meanwhile, BECFuzz reduces the number of edges to be instrumented, improving the efficiency of fuzzing. In order to effectively perform fuzzing, BECFuzz uses both edge coverage and coarse path coverage to record program states. The results of experiments show that BECFuzz discovers more paths than AFL, AFLFast and CollAFL. In addition to the capability of discovering more paths, BECFuzz uses smaller bitmaps compared to AFL, AFLFast and CollAFL. Meanwhile, BECFuzz exposes the most crashes and bugs in the experiments. It exposed 16 bugs during 12 hours while other fuzzers expose less than 14. Our contributions are as follows.

- We solve the problem of edge collision by directly instrumenting at edges of binaries. Therefore, we are able to detect bugs in SIoT applications because we do not need source codes.
- We show how to only instrument part of edges for efficient fuzzing. This decreases the memory overhead and increase the speed.
- We propose to perform fuzzing by using the coarse path coverage. With the help of this coverage, fuzzing can discover more paths and bugs.

This paper is organised as follows: Section 2 introduces the background of edge coverage-guided fuzzing. In the Section 3, we will explain how we design BECFuzz. Then, Section 4 shows the experimental results on 18 real-world programs. We introduce the related work in Section 5. Finally, we conclude this paper in Section 6.
Because AFL randomly assigns src and des to blocks, two different blocks may be assigned the same identifier, resulting in edge collision. Even though different blocks have different identifiers, Equation (1) may also calculate a same value. Based on this observation, CollAFL [13] extends Equation (1) to assure the unique value of each edge identifier:

\[ (\text{des} \gg x) \oplus (\text{src} \gg y) + z \]  

(2)

where \(x, y, z\) are the parameters determined during static analysis. Although CollAFL improves the performance of AFL, it neglects the invisible edges such as indirect calls. The missing indirect edges suggest that CollAFL still has the problem of edge collision.

The problem of collision leads fuzzing to neglect saving new seeds or unique crash inputs, which largely decreases the performance of fuzzing. For instance, in Fig.2(a), the edges \(BE\) and \(BD\) have the same edge identifier 3. Therefore, the edges \(BE\) and \(BD\) will be regarded as the same edge during fuzzing. In terms of path collision, the paths \(A \rightarrow B \rightarrow D \rightarrow G \rightarrow K\) and \(A \rightarrow B \rightarrow E \rightarrow G \rightarrow K\) collide because they have the same set of edge identifiers. Besides, if the paths \(A \rightarrow B \rightarrow D \rightarrow G \rightarrow K\) and \(A \rightarrow C \rightarrow E \rightarrow H \rightarrow K\) in Fig.2(a) have been exercised before, the new path \(A \rightarrow C \rightarrow E \rightarrow G \rightarrow K\) will be wrongly regarded as an old path during fuzzing because all the edge identifiers in the path exist. As a result, the new seed or unique crash input associated with the collision path will not be saved, reducing the chance to expose bugs.

All in all, the block-based edge identifiers do not resolve the collision thoroughly. An intuitive improvement is to instrument at edges and assign unique identifiers to each edge without block identifiers. However, instrumenting at edges is more complex than instrumenting at blocks [26], which impedes researchers to instrument at edges.

### 2.2 The Number of Edge Identifiers

Due to the block-based scheme, the existing ECGFs actually assign identifiers to all edges. However, the number of edge identifiers affects the performance of fuzzing. The edge identifiers are the indices of a bitmap, whose bytes indicate the states of edges, including being not examined, being examined, or being examined many times. Therefore, the size of bitmap is proportional to the number of edge identifiers. Because ECGFs inspect every byte in bitmap to determine new coverage and each examined edge will write
data to the bitmap, the size of bitmap affects the performance of fuzzing. When a program has a large number of edge identifiers, fuzzers have to increase the size of bitmap in order to avoid edge collision. As analysed in ColI AFL, increasing the size of bitmap will decrease the execution speed of fuzzing. Therefore, decreasing the size of bitmap is necessary for fuzzing.

As shown in Fig.2(a) and Fig.2(b), all edges in the CFG are assigned with identifiers, including conditional edges, unconditional edges, and edges without jumps. However, only part of the edges are related to inputs, and identifiers can be assigned only to those edges. In this paper, we will analyse which edges are assigned with identifiers.

2.3 The Overhead and Execution Speed of Fuzzing

2.3.1 The Run-time Overhead

Fuzzers instrument target programs statically [14], [15] or dynamically [16], [17] for coverage information. Static instrumentation cannot get the targets of indirect edges accurately, which may cause collision in fuzzing. On the other hand, dynamic instrumentation can accurately obtain the indirect edges. As for the execution speed, programs with dynamic instrumentation run much slower than programs with static instrumentation because of the run-time overhead of dynamic instrumentation. The execution speed is critical for fuzzing because a higher execution speed saves time for fuzzing to test more inputs, which has a higher chance to expose bugs. Because most existing fuzzers do not realise the problem of edge collision, they utilise static instrumentation to obtain coverage information. Therefore, in this paper we propose an approach to satisfy both high execution speed and avoiding edge collision.

2.3.2 The Memory Overhead

When saving a new seed or a new unique crash input, existing ECGFs only utilise the information of edge coverage. In ECGFs, more seeds or more crash inputs have a higher chance to expose more bugs, which suggests more coverage information may improve the performance of fuzzing. Path coverage tracks the order of edges, providing fuzzing with more coverage information. However, in practice, the run-time overhead and memory overhead of path coverage are too high to track all paths [13], [15]. Another solution is to use information of coarse path, i.e., paths without the order of edges, which has much lower overhead than the original path coverage. For example, many existing fuzzers [8–10], [20], [22], [23] calculate checksums for coarse paths to delete useless data in inputs. However, they do not use the checksum to save seeds or crash inputs. The reason is that the memory overhead of coarse path coverage is still high. Therefore, in this paper, we propose an approach to use the coarse path coverage information while reducing the memory overhead.

3 DESIGN OF BECFUZZ

Fig.1 shows how BECFuzz works. BECFuzz first instruments at specific edges to assign edge identifiers so that it differentiates edges. By only instrumenting at specific edges, BECFuzz saves space for bitmap and thus it increases the performance of fuzzing. To obtain more coverage information, BECFuzz utilises coarse path coverage (i.e., path coverage without the order of edges) to conduct fuzzing. Because the overhead of coarse path coverage is still high, BECFuzz rules out the major part of coarse paths to conduct fuzzing. When BECFuzz discovers a crash, it verify whether the crash is unique or not based on coarse path coverage.

3.1 Edge Identifiers

Edge identifiers are the coverage information in ECGFs, guiding fuzzing to save more seeds and crash inputs. The existing block-based scheme of edge identifier assignment causes edge collision, which is the motivation of BECFuzz to instrument at edges directly. With the implementation of edge instrumentation, BECFuzz can assign identifiers to edges without using block identifiers. BECFuzz utilises different schemes for direct edges and indirect edges. For direct edges, BECFuzz assigns identifiers to conditional edges; and BECFuzz assigns identifiers to edges that are indirect calls and indirect jumps.

3.1.1 Direct Edges

Direct edges are the control flow transfers where the target is encoded as immediate offset in instruction. Therefore, these edges can be obtained during static analysis, ensuring the ability of BECFuzz to assign identifiers to these edges with static instrumentation. As shown in Fig.2(c), BECFuzz only instrument at the conditional edges, which reduces the number of edge identifiers compared to Fig.2(b). The edges that are unconditional jumps or without any jump are not instrumented because they are always examined together with their predecessors. For example, in Fig.2(b), the unconditional edge $FJ$ is always examined when the conditional edge $CF$ is examined. Therefore, edges $FJ$ and $CF$ can be regarded as one edge, and only edge $CF$ need to be instrumented.
3.1.2 Indirect Edges
As indirect edges cannot be obtained during static analysis, BECFuzz assigns identifiers to them at run-time. Although it is hard to get the targets of indirect edges, the source blocks of indirect edges can be obtained during static analysis. BECFuzz statically instruments at the source blocks of indirect edges and the instrumented code obtains the targets of indirect edges at run-time. When the targets of indirect edges have not been examined as shown in Fig.3(a), BECFuzz does not assign identifiers to those edges. In Fig.3(b), when fuzzing has examined the indirect edge $AB$, BECFuzz assigns an identifier 51 to the edge. Then, the identifier 51 is used to conduct fuzzing. The edge $AC$ in Fig.3(c) is assigned identifier 52 the same way as edge $AB$.

3.2 Coarse Path Coverage
Existing fuzzers use the information of coarse path to remove useless data in inputs (e.g., AFL) or to count path frequency (e.g., AFLFast [20]). The information of coarse path is the path without the order of edges. For example, the path $A \to B \to E \to G \to K$ in Fig.2(a) has the set of identifiers $\{3, 5, 8, 15\}$ and fuzzers calculate a checksum of the set. Then, the value of the checksum $ck(\{3, 5, 8, 15\})$ is regarded as the coarse path identifier. If fuzzing uses the coarse path identifier to save seeds, it will get more seeds than only using edge identifier because new paths exist in examined edges. For instance, if paths $A \to B \to E \to G \to K$ and $A \to C \to E \to H \to K$ have been exercised, the set of examined edges will be $\{3, 5, 7, 8, 12, 15, 16, 21\}$. Then, the new path $A \to B \to E \to H \to K$ is considered as an old path because the edge identifiers $\{3, 5, 7, 16\}$ have existed. On the other hand, the checksum $ck(\{3, 5, 7, 16\})$ of the path has not been seen before, leading fuzzing to regard it as a new path.

However, it saves too many seeds and causes high memory overhead using coarse path identifier. For instance, in our experiments of application tcpdump, it gets about 2K seeds when using the edge identifier; however, it gets about 230K seeds when using the coarse path identifier. Therefore, BECFuzz will filter the result of coarse path coverage to keep the diversity of seeds, i.e., at least one edge in each execution path is new to other execution paths. BECFuzz utilises coarse path coverage to conduct fuzzing in three steps: 1) BECFuzz saves seeds if it discovers new edges; 2) if BECFuzz does not discover new edges, it utilises the coarse path identifier to determine new paths; 3) BECFuzz then utilises the edge identifiers to filter the result of step 2. Fig.4 shows when to save a seed in BECFuzz. The identifiers of the examined edges in Fig.4(a) are $\{2, 4, 6, 8, 10, 12\}$, and the identifiers of the examined edges in Fig.4(b) are $\{1, 3, 5, 7, 9, 11\}$. These two executions both discover new edges, and all the edges are examined by the two executions. In Fig.4(c) and Fig.4(d), although fuzzing does not discover new edges, it discovers new paths based on coarse path identifiers. We call the new paths whose edges have all been examined before as NAOE paths (new paths among old edges). If an input that exercises a NAOE path, and at least one of the edges in the NAOE path cannot be found in other NAOE paths, then we will save the input as a new seed. In the third and fourth executions, some edges have not been seen in other NAOE paths, thus BECFuzz saves the seeds. In Fig.4(e), although fuzzing discovers a NAOE path, all the edges examined in the fifth execution have already
been seen in other NAOE paths, \textit{i.e.}, NAOE paths from the third and fourth executions. Therefore, in the fifth execution, BECFuzz does not save any seed.

To save crash inputs, because BECFuzz wants to analyse each unique crash input, it saves all new crash inputs based on coarse path identifiers. Similar to seeds, the coarse path solution will save more crash inputs than the edge solution. With more unique crash inputs, fuzzing has a higher chance to expose more bugs.

### 4 Evaluation

We run our experiments on the machine with CPU AMD Ryzen Threadripper 2990WX 32-Core Processor and five fuzzers are evaluated \textit{i.e.}, AFL [8], AFLFast [20], CollAFL [13], BECFuzz-AFL and BECFuzz-AFLFast. Each fuzzer runs on one core of CPU and the timeout is 12 hours. The 18 target programs are shown in Table 1. The third column of the table is the number of edges, the fourth is the number of conditional edges, and the fifth is the number of indirect edges. We choose these applications because they are from different fields, such as image and PDF files, and some of them could be used in SIoT applications. Meanwhile, many SIoT applications can only be accessed by binaries. Note that we evaluate AFL, AFLFast and CollAFL in our experiments because other coverage-guided fuzzers do not focus on the problem of edge collision. Our solution is orthogonal to other fuzzers, such as Driller [9] and Angora [25]. To evaluate on binaries, AFL and AFLFast run in the QEMU mode. Besides, to avoid cycle explosion [27], we add a lower bound to AFLFast. Because we cannot get the source code of CollAFL, we implement it by utilising Dyninst and assign a unique identifier to each edge.

#### 4.1 The Size of Bitmap

Our BECFuzz intends to reduce the size of bitmap while avoiding edge collision. A smaller size of bitmap indicates that fuzzing uses less space in memory. Therefore, BECFuzz can reduce the memory overhead during fuzzing. In Table 1, the sixth column shows the size of bitmap used in the experiments for each target program. All the five fuzzers use the same size of bitmap, which is determined by BECFuzz. For instance, the bitmap size of ‘\texttt{objdump}’ application is 64KB because the overall number of conditional edges and indirect edges is less than 64KB. However, the number of all edges of \texttt{objdump} is larger than 64KB, which suggests edge collision exists in AFL, AFLFast and CollAFL because they assign identifiers to all edges. In order to avoid edge collision, they have to enlarge the size of bitmap, which reduces the execution speed of fuzzing [13].

During fuzzing, BECFuzz uses less edge identifiers to exercise more paths. Fig.5 shows the average percentage of bitmap used by fuzzers during five trials. Due to the same size of bitmap, the percentage in Fig.5 suggests the number of edge identifiers used in the exercised paths. According to Fig.5, BECFuzz-AFL and BECFuzz-AFLFast have the lowest percentage in all the 18 target programs. For instance, the percentage of bitmap used by AFL is four times of the percentage used by BECFuzz-AFL. Our BECFuzz successfully and largely reduces the number of edge identifiers during fuzzing, which can increase the execution speed.

Because we implement CollAFL via assigning identifiers to edges directly, it may miss some edges in target programs, such as edges related to signals. In such cases, the percentage of bitmap used by CollAFL is less than AFL or AFLFast, such as \texttt{bison} in Fig.5. However, theoretically, CollAFL uses more edge identifiers than AFL because CollAFL solves the problem of edge collision. For instance, CollAFL occupies more space of bitmap than AFL in the target program \texttt{readelf} in Fig.5.

#### 4.2 New-edge Seeds (Paths)

During fuzzing, a seed indicates that one execution path has been exercised. The existing ECGFs save seeds based on edge coverage, \textit{i.e.}, the edge bitmap. BECFuzz saves seeds based on two different kinds of coverage information, \textit{i.e.}, the edge coverage and the coarse path coverage. Fig.6 shows the average number of seeds saved based on new edges. New-edge seeds are saved only when fuzzing discovers new edges, and the new edges are determined based on the edge bitmap. Therefore, the performance of new-edge paths indicates the ability to resolve edge collision. According to Fig.6, BECFuzz-AFL and BECFuzz-AFLFast discover more new-edge paths than other fuzzers on almost all the 18 programs. Meanwhile, BECFuzz-AFL and BECFuzz-AFLFast discover new-edge paths faster than other fuzzers. For instance, after 12 hours, BECFuzz-AFL discovers 191.54\% more new-edge paths than AFL on the application \texttt{readelf}. On average, BECFuzz-AFL discovers...
Fig. 5. The average percentage of edge bitmap used by fuzzers in five trials. BECFuzz-AFL and BECFuzz-AFLFast occupy the least space of edge bitmap, which indicates BECFuzz successfully reduces the number of edge identifiers.

65.72% more new-edge paths than AFL. BECFuzz-AFLFast discovers more new-edge paths than AFLFast on all the 18 applications. For example, BECFuzz-AFLFast discovers 76.11% more new-edge paths than AFLFast on application gm. On average, BECFuzz-AFLFast discovers 61.2% more new-edge paths than AFLFast. BECFuzz performs better than CollAFL in terms of path discovery because BECFuzz solves the problem of edge collision on indirect edges. According to Fig.6, among all the 18 programs, BECFuzz-AFL discovers more new-edge paths than CollAFL on 15 programs. On average, BECFuzz-AFL discovers 9.13% more new-edge paths. Meanwhile, BECFuzz-AFLFast discovers more new-edge paths than CollAFL on 17 programs. BECFuzz-AFLFast discovers 17.21% more new-edge paths than CollAFL on average.

During fuzzing, all five fuzzers save new-edge seeds based on the edge bitmap, which is the same size for all five fuzzers. Therefore, according to Fig.5 and Fig.6, we conclude that BECFuzz can discover more new-edge paths than AFL, AFLFast and CollAFL while it uses less space of bitmap. It indicates that BECFuzz can instrument less edges while performing better in terms of path discovery.

4.3 Crashes and Bugs

Fig.7 shows the average crashes which are exposed during fuzzing. We expose crashes in five programs, i.e., bison, cflow, lou_translate, pspp, and sfconvert. After 12 hours, BECFuzz-AFL exposes 0.2, 7.0, 3.2 more crashes on average than AFL on bison, lou_translate and pspp, respectively. On the other hand, BECFuzz-AFLFast exposes 7.2, 6, 3.8 and 4 more crashes than AFLFast on cflow, lou_translate, pspp and sfconvert, respectively. For all the five programs in Fig.7, AFL, AFLFast, BECFuzz-AFL, BECFuzz-AFLFast and CollAFL exposes 34.6, 36, 41.4, 57, and 43.6 crashes, respectively. BECFuzz-AFLFast performs the best in terms of crash discovery. On the other hand, Fig.7 indicates that BECFuzz exposes crashes faster than other fuzzers. For instance, on program lou_translate, BECFuzz-AFL discovers 7.73 crashes in six hours while AFL only exposes 1.2 crashes in the same time.

We then use afl-collect [28] to determine the unique crashes of the programs. It utilises the GDB plugin exploitable [29] to de-duplicate crashes. We further manually analyse the unique crashes and confirm bugs. Table 2 shows the number of bugs found during fuzzing. BECFuzz-AFLFast, which discovers the most bugs, exposes 16 bugs in 12 hours. BEC-AFL and CollAFL exposes the second most bugs, which are 14 bugs. Table 2 indicates that BECFuzz is effective on exposing bugs. Other fuzzers may miss some bugs during fuzzing. For example, on the program bison, AFL and CollAFL misses a bug that causes memory error. AFL and AFLFast discovers zero bugs on program pspp while others expose more than two bugs.
Some fuzzers [9], [12], [30]–[33] utilise symbolic execution to perform dynamic analysis on magic values, checksums, or hashes) in target programs. Coverage-guided fuzzing utilises the coverage information to select and mutate test cases. To differentiate the edges of a program, the widely-used fuzzer AFL [8] uses a bitmap to trace edges. Many fuzzers [5], [9], [10], [12], [13], [20]–[22] are implemented based on AFL to improve the performance of AFL and effectively expose bugs in programs. AFLFast [20] improves the strategy of seed selection for AFL and speeds up the ability of path discovery. MOPT-AFL [10] designs a new mutation strategy for AFL and improves the effectiveness of mutation. CollAFL [13] uses extended hash functions to resolve the problem of edge collision of AFL. However, they do not recognise the problem of reducing the size of bitmap. AFLGo [5] improves the strategy of seed selection and mutation times of AFL, which aims to reach the pre-determined location of target programs. Driller [9] uses symbolic execution to help AFL solve path constraints.

One of the most important challenges for coverage-guided fuzzing is to resolve path constraints (e.g., checks on magic values, checksums, or hashes) in target programs. Some fuzzers [9], [12], [30]–[33] utilise symbolic execution to

---

**Table 2**

<table>
<thead>
<tr>
<th>Applications</th>
<th>BECFuzz-AFL</th>
<th>BECFuzz-AFLFast</th>
<th>AFL</th>
<th>AFLFast</th>
<th>CollAFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>bison</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cflow</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lou_translate</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pspp</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sfconvert</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>14</td>
<td>16</td>
<td>12</td>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

---

Fig. 6. The average number of new-edge seeds (paths). All five fuzzers save seeds while discovering new edges. BECFuzz discovers more paths than the other fuzzers.

Fig. 7. The average number of crashes. BECFuzz exposes crashes faster than other fuzzers.

---

5 Related Work

Coverage-guided fuzzing utilises the coverage information to guide the mutation of test cases. To differentiate the edges of a program, the widely-used fuzzer AFL [8] uses a bitmap to trace edges. Many fuzzers [5], [9], [10], [12], [13], [20]–[22]
help fuzzing resolve path constraints and reach more code coverage. REDQUEEN [23] resolves part of the path constraints based on the observation that part of the inputs can directly influence the program state. T-Fuzz [24] removes the path constraints in target programs to obtain more seeds, which are then tested by the original target programs. Some fuzzers [34]–[36] use taint analysis to help fuzzing reach more code coverage.

To fuzz on binaries, fuzzers utilise different tools to trace the coverage. AFL-Dyninst [14] uses Dyninst to statically instrument at all blocks to trace the edge coverage. On the other hand, AFL-Pin [17] dynamically instruments target programs by using Intel Pin [37]. AFL-QEMU traces coverage without instrumentation but changes the emulator QEMU to get the information about coverage. To get the information about CFGs of target programs, VUzzer [16] uses both static and dynamic tools, i.e., IDA Pro [38] and Intel Pin. To remove the path constraints in target programs, T-Fuzz transforms the original programs with tools angr [39] and radare2 [40]. TaintScope [41] dynamically instruments target programs with Intel Pin, which aims to accomplish taint analysis. With the help of Dyninst, UnTracer [15] instruments at blocks to trace coverage at full speed.

6 Conclusion

In this paper, we focus on fuzzing binaries because in many situations, like SloT applications, we can only have access to the binaries. We propose BECFuzz to reduce the overhead, and to improve the effectiveness when fuzzing binaries. We argue that only conditional and indirect edges need to be instrumented, which largely reduce memory overhead. Meanwhile, BECFuzz proposes the idea of instrumenting directly at edges, which avoids the problem of edge collision. To perform fuzzing more effectively, BECFuzz uses the information of both edge coverage and coarse path coverage. To reduce the overhead of coarse path coverage, BECFuzz ignores most of the seeds saved by coarse path coverage. The results of experiments demonstrate the effectiveness and efficiency of BECFuzz. It discovers more paths than state-of-art fuzzers while using less bitmap space. Meanwhile, BECFuzz executes target programs faster and spend less memory than AFL, AFLFast and CollAFL. As a result, BECFuzz discovers crashes faster and exposes more bugs than current fuzzers.

References

Xiaogang Zhu is a PhD student of Computer Science and Engineering at the Swinburne University of Technology. He received his B.Eng degree from Xi’an Jiaotong University in 2012 and M.Eng degree from Xi’an Jiaotong University in 2015. Currently, his research is about searching vulnerabilities in programs. He’s interested in detecting techniques such as fuzzing, machine learning and symbolic execution. He has published papers on top journals, such as IEEE Transactions on Dependable and Secure Computing (TDSC), and conferences such as ACM ASIA Conference on Computer and Communications Security (AsiaCCS).

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

Allireza Jolfaei received the Ph.D. degree in Applied Cryptography from Griffith University, Gold Coast, Australia. He is the Program Leader of Master of IT in Cyber Security in the Department of Computing at Macquarie University, Sydney, Australia. His current research areas include Cyber and Cyber-Physical Systems Security. He has authored over 100 peer-reviewed articles on topics related to cybersecurity. He has served over 10 conferences in leadership capacities including program co-Chair, track Chair, session Chair, TPC Chair, and Technical Program Committee member, including IEEE TrustCom and IEEE INFOCOM. He is a Senior Member of the IEEE and a Distinguished Speaker of the ACM on the topic of Cyber Security.

Mohammad Sayad Haghighi (IEEE SM'18) received the Ph.D. degree in telecommunication systems from K. N. Toosi University of Technology, Tehran, Iran. He is the Head of IT Department at the School of Electrical and Computer Engineering, University of Tehran, Iran. Prior to joining the University of Tehran, he was an Assistant Professor at Iran Telecom Research Center. Since 2009, he has been holding research positions with Australian universities. His research interests include wireless ad hoc networks as well as cybersecurity. Dr. Sayad Haghighi has served as a Program Committee Member of many conferences such as IEEE WNS, IEEE SICK, IEEE HPCC, IEEE DASC, and IEEE LCN. He has won several national grants including some from Iran National Science Foundation.

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

Yang Xiang received his PhD in Computer Science from Deakin University, Australia. He is a Professor and the Dean of Digital Research & Innovation Capability Platform at Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. In particular, he is currently leading his team developing active defense systems against large-scale distributed network attacks. He is the Chief Investigator of several projects in network and system security, funded by the Australian Research Council (ARC). He has published more than 200 research papers in many international journals and conferences, such as IEEE TC, IEEE TPDS, IEEE TIFS, and IEEE JSAC.