석 사 학 위 논 문
Master's Thesis

함수 관련도를 이용한 퍼징의 커버리지 향상

Function Relevance based Fuzzing for Coverage Improvement

2021

이 아 청 (李 我 淸 Lee, Ahcheong)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

함수 관련도를 이용한 퍼징의 커버리지 향상

2021

이 아 청

한 국 과 학 기 술 원

전산학부

함수 관련도를 이용한 퍼징의 커버리지 향상

이 아 청

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 6월 10일

심사위원장 김 문 주 (인)

심 사 위 원 유 신 (인)

심 사 위 원 강 지 훈 (인)

Function Relevance based Fuzzing for Coverage Improvement

Ahcheong Lee

Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
June 10, 2021

Approved by

Moonzoo Kim
Professor of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS 이아청. 함수 관련도를 이용한 퍼징의 커버리지 향상. 전산학부 . 2021년. 30+iv 쪽. 지도교수: 김문주. (영문 논문)
Ahcheong Lee. Function Relevance based Fuzzing for Coverage Improvement. School of Computing . 2021. 30+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

테스트 자동 생성 기술인 퍼징은, 크고 복잡한 프로그램에서도 오류를 검출하는데 높은 성능을 보여, 널리 이용되고 있다. 하지만, 테스트 대상 프로그램의 시맨틱 정보를 이용하면 퍼징의 테스트 커버리지 성능을 높일 수 있는 잠재력이 있음에도, 이러한 시맨틱 정보를 이용하는데 필요한 실행비용이 높고, 기술적인 난이도가 높아 이러한 시맨틱 정보를 사용하는 퍼징 기술에 대한 연구는 더딘 편이다. 이러한 한계를 극복하기 위해 본 논문에서는 동적 함수 관련도를 사용하는 첫 퍼저인 FRIEND를 제시한다. FRIEND는 타겟 분기 b_t를 포함한 함수 f_t에 대해 높은 연관도를 가지는 함수를 식별하여 이 정보를 이용하여 어떤 테스트 입력을 변이할지, 테스트 입력의 어떤 바이트를 변이할 지를 결정하여 효율적으로 높은 커버리지와 버그 식별 능력을 가지는 테스트를 생성하도록 한다. FRIEND를 LAVA-M의 4개의 프로그램과 10개의 실제로 활발히 사용되는 프로그램에 적용한 결과, 다른 최신 퍼저 (AFLFast, Angora, FairFuzz, RedQueen)과 비교하여 높은 커버리지와 버그 식별 능력을 보였다.

핵 심 낱 말 소프트웨어 테스팅, 자동화 테스트 생성 기술, 퍼징, 함수 관련도

Abstract

Fuzzing has become popular as a software bug detection technique for its high bug detection ability (covering large execution space of a complex target program fast). Although semantic information of a target program can be useful to improve test coverage of fuzzing, still most fuzzers do not utilize valuable semantic information of a target program much. This is because obtaining such semantic information is technically difficult and/or costly (i.e., causing high runtime overhead). To resolve such limitation, this dissertation proposes FRIEND, which is the first fuzzer to use "dynamic function relevance", which is a salient method to improve test coverage and crash bug detection ability cost-effectively. FRIEND identifies functions closely relevant to a target function f_t containing a target branch b_t and utilizes this information to select test inputs and input bytes to mutate. I found that the dynamic function relevance metric is simple and cheap to calculate and can improve fuzzing performance significantly. I have applied FRIEND to 4 LAVA-M benchmark programs and 10 popular real-world programs. The experiment results demonstrated that FRIEND covers significantly more execution paths and detects more crashes than other cutting-edge fuzzers (i.e., AFLFast, Angora, FairFuzz, and RedQueen).

Keywords Software Testing, Automated test generation, fuzzing, function relevance

Contents

List of Tables

List of Figures

Chapter 1. Introduction

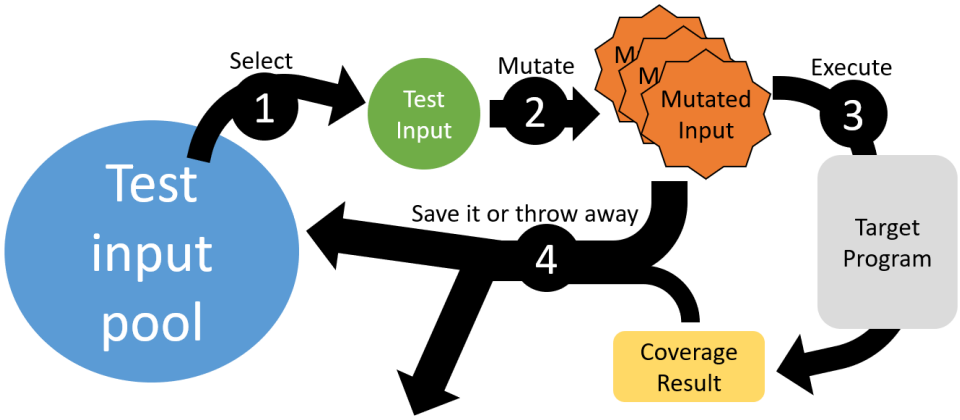

Figure 1.1: Coverage guided fuzzing process

Fuzz testing (a.k.a. fuzzing) has become popular as a software bug detection technique for its high bug detection ability. This is because fuzzing can cover large execution space of a complex real-world software program fast. Unlike sophisticated automated testing techniques (e.g., symbolic execution [6, 20, 3] and concolic testing [32, 19]), fuzzing generates diverse test inputs fast by directly mutating test input bytes with various heuristics to improve test coverage [29]. Thus, major software companies like Google have applied fuzzing techniques for general quality assurance (e.g., [30, 13]).

1.1 Research Background

To improve test coverage and crash bug detection of a target program P, a fuzzer can utilize *semantic information* of a target program P. For example, suppose that, to cover a target branch b_t in a function f_t, functions g and h have to be executed before f_t. A smart fuzzer can utilize this valuable semantic information to generate test inputs to cover b_t by assigning a high weight to a test input that executes g and h before f_t. However, most fuzzers do not use semantic information of P because extracting such semantic information of P is technically challenging (i.e., both static time code analysis and run-time dynamic analysis are required) and it causes high runtime overhead to calculate complex semantic information.

1.2 Fuzzing Background

Fuzzing is an automatic test input generation technique that gernerates new test inputs by mutating existing test inputs. Typically, fuzzing techniques treats a test input as a simple sequence of bytes and mutates each byte to make test inputs that can cover new code area or trigger defects in a program under test.

Figure 1.1 shows a common process of coverage guided fuzzing techniques. covearage guided fuzzers are fuzzing techniques that utilizes coverage result of test input executions to generate as many test inputs as possible that can cover large coverage area of a target program.

1. First, the fuzzer selects a test input to mutate.

2. Second, the fuzzer mutates the selected test input to generate multiple new test inputs. For efficiency, typically a fuzzer evaluates each selected test input to determine how many inputs to generate. (i.e. The fuzzer determines how much time to spend on mutating a selected test input.) How fuzzers evaluate test inputs is described in section 4 (Related works).

3. Third, the fuzzer executes the new test inputs on the target program to obtain dynamic execution data.

4. Lastly, the fuzzer determines whether to keep the generated new test input or not. Generally, most fuzzers save test inputs that can cover new path coverage or trigger new crashes.

It is important to select a good test input to mutate that can be mutated to test inputs that show diverse program behaviors. And, it is also important to select proper test input bytes to mutate because the search space of new mutated test inputs is combinatorial complexity, so if a fuzzer selects too many input bytes, it is hard to generate good test inputs in limited time slot.

1.3 Thesis Statement and Contributions

1.3.1 Thesis Statement

The thesis statement of this dissertation is as follows:

Function relevance based input selection and input byte selection can improve the effectiveness of coverage guided fuzzing in terms of coverage and bug detection power.

1.3.2 Proposed Approach

As I metioned above, current state-of-the-art fuzzers do not use semantic information of target program under test because extracting and utilizing semantinc information of program require high runtime overhead. To resolve such limitation of current fuzzing techniques, I have developed FRIEND (Function Relevance semantIcs basEd fuzziNg methoD), which is a novel fuzzing technique to adopt "dynamic function relevance metric" [19]. To cover a target branch b_t in a function f_t, FRIEND identifies functions closely relevant to f_t (saying g, h) and utilizes this semantic information to select test inputs (Section 2.4) and input bytes to mutate (Section 2.5). In more detail, initially FRIEND uses dynamic taint analysis (DTA) to identify and mutate an input byte i_m that affects the branch condition of b_t (i.e., the branch condition has data-dependency on the input byte i_m). Then, for b_t that is not covered by DTA-based test input generation, FRIEND generates test inputs that can cover g and h which are closely relevant to f_t; consequently, test inputs mutated/generated from such test inputs will likely cover b_t in f_t. I have applied FRIEND to 14 target programs (4 LAVA-M programs and 10 real-world programs) and demonstrated that FRIEND covers 25.8% to 513.6% relatively more unique execution paths and detected 10 to 20 more crashes than cutting-edge fuzzers such as AFLFast [5], Angora [7], FairFuzz [25] and RedQueen [2].

1.3.3 Contributions

The main contributions of this paper are as follows:

1. I have developed FRIEND that is the first fuzzer to apply *dynamic function relevance metric*, which is a salient method to improve test coverage and crash bug detection ability cost-effectively.
2. I have performed a series of the experiments where I have empirically evaluate FRIEND and other cutting-edge fuzzers (i.e., AFLFast, Angora, FairFuzz, and RedQueen) and demonstrated that FRIEND achieves significantly higher coverage and detects more crash bugs than the cutting-edge fuzzers.
3. I have detected 21 new crash bugs by using FRIEND and reported those bugs to the original developers to improve the quality of the open source target programs.

1.4 Structure of Dissertation

The remainder of this dissertation is structured as folows. Chapter 2 presents FRIEND with its motivating example. Chapter 3 presents the empricial evaluation setting and results of FRIEND, chapter 4 describes realted works of FRIEND, and chapter 5 concludes the dissertation with future works.

Chapter 2. FRIEND: Function Relevance semantIcs basEd fuzziNg methoD

2.1 Motivating Example

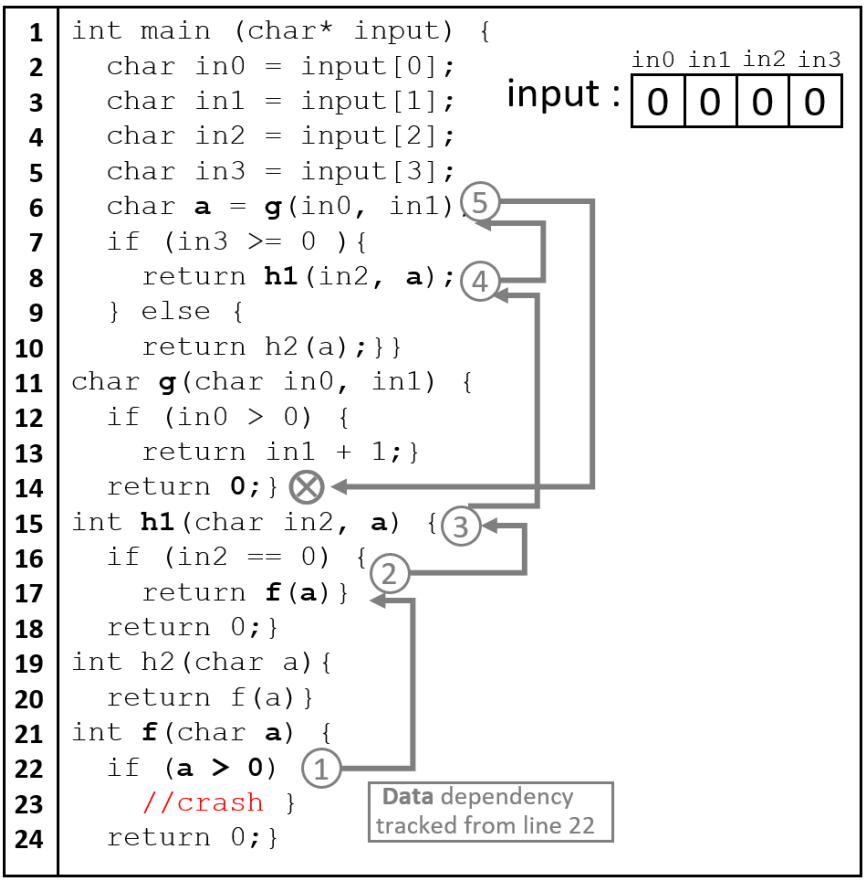

Figure 2.1: DTA Fail example

2.1.1 Example Description

Figure 2.1 shows an example code and data depedency tracked from a target branch b_{23} (line 23) with 4-bytes length input 0000. In this case, DTA can not identify input bytes that affect b_{23} because there is no data dependency between b_{23} and the given test input. Therefore, a fuzzer can not specify which input bytes to mutate using DTA in this example to cover a target branch b_{23}.

Figure 2.2 shows the same example code and data dependency tracked from a branch b_{12} (line 12). The branch b_{12} has data dependency with the first input byte of a test input 0000. Let's assume that function f and function g has high function relevance. (i.e. function f and g are highly related to each other.) Since f and g are highly related, if we select and mutate the first input byte 0 to 9 and generate an input 9000, we can cover the target branch b_{23}, and cover the crashed line 30.

Figure 2.3 shows a function relevance table of functions with the function f, and Figure 2.1.1 shows a control flow graph showing execution paths of the test inputs 0000 and 9000.

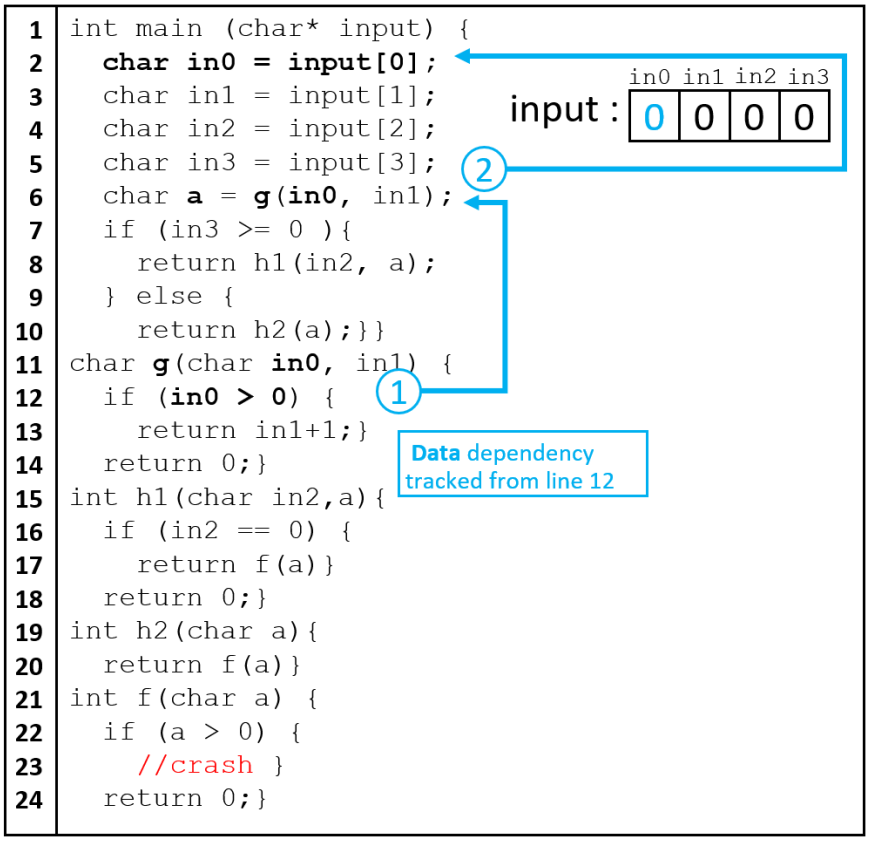

Figure 2.2: Same example code with data dependency tracked from line 12

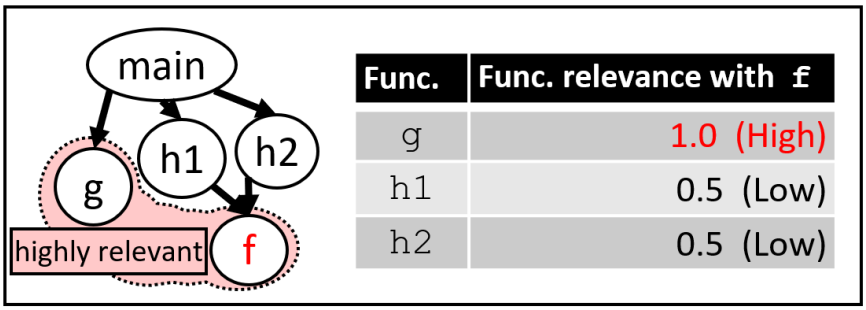

Figure 2.3: Function relevance table of function f

Although function f and function g have no direct call relation, it is safe to assume that the functions f and g are highly related to each other, while the relavance between function f and functions $h1, h2$ are not, since when the function f is executed, the function g is always executed but the functions $h1$ and $h2$ are not.

To generate a test input that covers a target branch b_{23} in `f` (at line 23) in Figure 2.1, suppose that a fuzzer performs dynamic taint analysis (DTA) as follows. Suppose that a fuzzer executes the target program with an initial 4-byte long input 0000; its execution path is shown as a dotted gray line in Figure 2.1.1 and its DTA result is shown in Figure 2.1 (see arrows with labels ①, ②, ③, ④, and ⑤ in order). DTA fails to identify input bytes to mutate to cover b_{23} because the branch condition (i.e., `a > 0` at line 22) of b_{23} does not have data-dependency on any input bytes (Figure 2.1). Thus, a DTA based fuzzer fails to cover b_{23}.

If a fuzzer naively analyzes both data dependency and control dependency together, it will find that

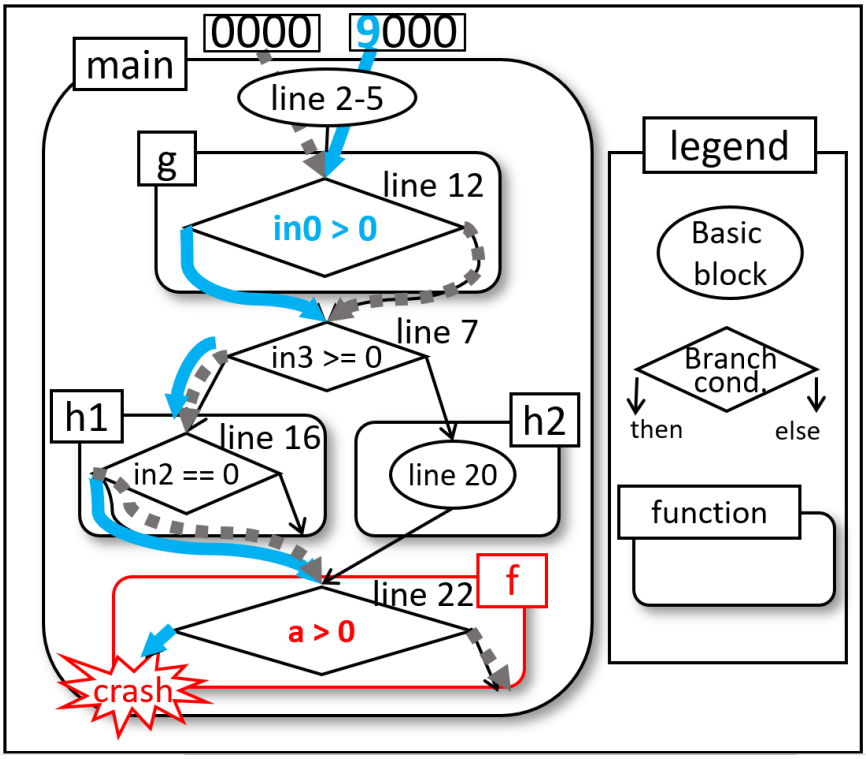

Figure 2.4: Control flow graph and execution paths of input 0000 and 9000

b_{23} depends on `input[0]`, `input[2]`, and `input[3]` and mutate all of them (i.e., 75% of all input bytes). As a result, by mutating too many input bytes, it will increase execution search space unnecessarily large and decrease the effectiveness and efficiency of fuzzing.

2.2 Overall Process

Algorithm 1 shows the overall process of FRIEND. It receives a target program $Prog$ and a set of initial seed test inputs $INIT$, and returns a set of generated test input TI. FRIEND uses function relevance (Section 2.3) to select test inputs (Section 2.4) and input bytes to mutate (Section 2.5). We highlighted the important steps in Algorithm 1 to which function relevance metric is applied (i.e., lines 12, 16, 17, and 29).

First, FRIEND executes initial seed test inputs $INIT$(lines 5–13). For each seed test input, it obtains a corresponding execution *path*, a taint information *input_taint*, and a function call trace $func\_trace$ (line 7). Also, it gets uncovered target branches *target_br* (lines 9–11) each of which is *adjacent* to the execution paths of $INIT$. For example of Figure 2.1, a target branch at line 30 is adjacent to the grey dotted execution path of the test input 0000 (see Figure 2.3). In addition, FRIEND updates function relevance information between every pair of the functions executed so far by using $func\_trace$ (line 12, see Definition 1 for the detail).

Then, FRIEND repeats the following steps until it covers all target branches or a given time budget is completely consumed (lines 14–31).

Algorithm 1: the overall process of FRIEND

Input: $Prog$: a target program, $INIT$: a set of initial seed test inputs

Output: TI : a set of generated test inputs

1 $target\_br \leftarrow \emptyset$: a set of uncovered branches adjacent to the explored paths
2 $FR \leftarrow$ empty map : a map from a pair of functions to the relevance between the functions
3 $Taints \leftarrow$ empty map : a map from a test input to taint information
4 $TI \leftarrow INIT$
5 **foreach** $input \in INIT$ **do**
6 //Get the path, tainted input bytes of executed branches, and function execution trace
7 $path, input\_taint, func\_trace \leftarrow$ Run $Prog(input)$
8 $Taints[input] \leftarrow input\_taint$
9 **foreach** *uncovered branch b that is adjacent to path* **do**
10 $target\_br \leftarrow target\_br \cup \{b\}$
11 **end**
12 $FR.update(func\_trace)$
13 **end**
14 **while** $target\_br \neq \emptyset$ and timeout is not reached **do**
15 Select a branch b_t from $target\_br$
16 $ti \leftarrow$ `SelectTestInput`(b_t, TI, FR)
17 $bytes \leftarrow$ `SelectBytes`$(b_t, Taints[ti], FR)$
18 **while** b_t is not covered and timeout for b_t is not reached **do**
19 $new\_ti \leftarrow mutate(ti, bytes)$
20 $TI.append(new\_ti)$
21 $path, input\_taint, func\_trace \leftarrow$ Run $Prog(new\_ti)$
22 $Taints[new\_ti] \leftarrow input\_taint$
23 **foreach** an adjacent uncovered branch b' of $path$ **do**
24 $target\_br \leftarrow target\_br \cup \{b'\}$
25 **end**
26 **foreach** a branch b'' newly covered by $path$ **do**
27 $target\_br \leftarrow target\_br - \{b''\}$
28 **end**
29 $FR.update(func\_trace)$
30 **end**
31 **end**

1. FRIEND selects a target branch b_t from $target\_br$ (line 15) [1] and selects a test input ti in a test input pool TI to cover b_t (line 16, Algorithm 2 in Section 2.4).

2. It selects *bytes* which are input bytes of ti that affect the branch condition of b_t by using DTA and function relevance (line 17, Algorithm 3 in Section 2.5).

[1] The same branch b_t can be selected multiple times in the `while` loop (lines 14–31).

3. It repeats the following steps until b_t is covered or a given time budget for b_t is completely consumed (lines 18–30):

   (a) FRIEND generates a new test input *new_ti* by mutating the selected input bytes *bytes* (line 19). Then, it adds *new_ti* to TI (line 20), executes *new_ti* (line 21), and records taint information (line 22).

   (b) It updates *target_br* by adding new uncovered branches adjacent to *path* of *new_ti* (lines 23–25) and by removing target branches covered by *path* of *new_ti* (lines 26–28).

   (c) It updates function relevance information by considering the function call trace of *path* (line 29)

2.3 Definition of Function Relevance

Among the dozens of function relevance/coupling metrics (e.g., [10, 26, 11, 23, 17, 1]), FRIEND decides to use *dynamic function relevance metric* for its intuitive characteristics and its very low runtime cost to calculate (dynamic function relevance was originally proposed to reduce false alarms of unit testing [19]). FRIEND defines and applies dynamic function relevance as follows:

Definition 1. Let TI be a set of generated test inputs with unique path coverage. The function relevance $FR(f, g)$ between two functions f and g is defined as :

$$FR(f,g) = \frac{|\{ti \in TI |\ ti \text{ that executes both } f \text{ and } g\}|}{|\{ti \in TI |\ ti \text{ that executes } f\}|} \in [0,1]$$

A function g is *highly relevant* to f if $FR(f, g)$ is high (i.e., $FR(f, g) > \tau$ where τ is a user-given threshold). Intuitively speaking, high $FR(f, g)$ means that f and g are frequently executed together and it means that f may have high dependency on g. Thus, $FR(f, g)$ can be used to identify test inputs that cover functions highly relevant to f and likely cover a target branch in f (Section 2.4). Note that the runtime overhead to calculate $FR(f, g)$ is negligible, because $FR(f, g)$ is calculated based on function call traces and counting the number of function call traces that contain f or both f and g is very simple (Section 3.2.5).

2.3.1 Computing Dependency of a Target Function on Other Functions [2]

Suppose that a program has a target function f and other function g and it has n_f system test executions that invokes f. Based on function call profiles, we compute *dependency* of f on g as $p(g|f)$. Given a static call graph $G(V, E)$ (see Def. 2) and system test executions, we compute $p(g|f)$ as follows:

- Case 1: for g which is a *predecessor* of f in $G(V, E)$, $p(g|f)$ is calculated as $\frac{n_1}{n_f}$ where n_1 is a number of system executions where g calls f directly or transitively.
- Case 2: for g which is a *successor* of f in $G(V, E)$, $p(g|f)$ is calculated as $\frac{n_2}{n_f}$ where n_2 is a number of system executions where f calls g directly or transitively.

[2] This section is quoted from Yunho Kim at el. [19] to supplement enough explanation of the concept of function relevance. The use is authorized by the original author.

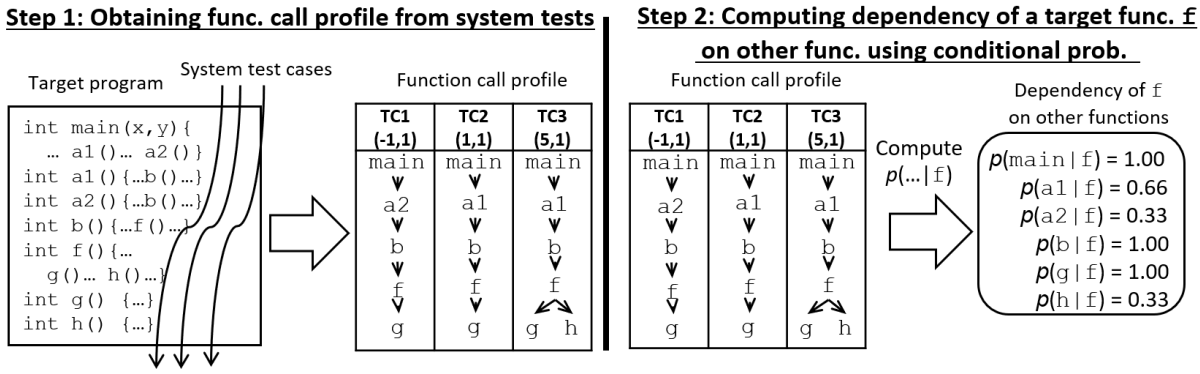

Figure 2.5: Computing function relevance example

- Case 3: for g which is a *successor* and *predecessor* of f in $G(V, E)$ (i.e., there exists a recursive call cycle between f and g), $p(g|f)$ is calculated as $\frac{n_3}{n_f}$ where n_3 is a number of system executions where f calls g or g calls f directly or transitively.

Definition 2. A *static call graph* $G(V, E)$ is a directed graph where V is a set of nodes representing functions in a program and E is a relation $V \times V$. Each edge $(a, b) \in E$ indicates that a directly calls b. We call a node p as a *predecessor* of f if there exists a path from p to f. We call a node s as a *successor* of f if there exists a path from f to s.

For example, Step 1 of Fig. 2.5 shows three test cases (-1,1), (1,1), and (5,1) and their corresponding function call profiles for the program. Based on the profiles, we calculated dependency of f on other functions as follows:

- $p(\texttt{main}|\texttt{f}) = 1.00\ (= \frac{n_1}{n_f} = \frac{3}{3})$
- $p(\texttt{a1}|\texttt{f}) = 0.66\ (= \frac{n_1}{n_f} = \frac{2}{3})$
- $p(\texttt{a2}|\texttt{f}) = 0.33\ (= \frac{n_1}{n_f} = \frac{1}{3})$
- $p(\texttt{b}|\texttt{f}) = 1.00\ (= \frac{n_1}{n_f} = \frac{3}{3})$
- $p(\texttt{g}|\texttt{f}) = 1.00\ (= \frac{n_2}{n_f} = \frac{3}{3})$
- $p(\texttt{h}|\texttt{f}) = 0.33\ (= \frac{n_2}{n_f} = \frac{1}{3})$

2.4 Selection of Test Input to Mutate

Algorithm 2 shows how FRIEND selects a test input to mutate to cover a target branch b_t. FRIEND selects a test input that (1) reaches the branch condition of a target branch b_t, (2) has not been selected to target b_t before, and (3) has the highest function relevance score with regard to b_t among the test inputs that has not been selected before. We define a function relevance score $TI_{score}(ti, b_t)$ of a test input ti with regard to b_t as follows:

Definition 3. The function relevance score $TI_{score}(ti, b_t)$ of ti with respect to a target branch b_t in f_t is defined as follow:

$$TI_{Score}(ti, b_t) = \frac{|\{g \mid g \text{ is executed by } ti \text{ and } FR(f_t, g) > \tau\}|}{|\{h \mid h \text{ is executed by } ti\}|}$$

where τ is a user-given threshold.

Algorithm 2: Test input selection algorithm

1 **Function** `SelectTestInput`(b_t, TI, FR)**:**

2 $ti_{sel} \leftarrow ti \in TI$ such that

1. reaches the branch condition of b_t, and
2. has not been selected to target b_t before, and
3. has the highest function relevance score $TI_{Score}(ti, b_t)$ among the unselected test inputs

return ti_{sel}

Our conjecture is as follows. A test input ti that has high function relevance score with respect to a target branch b_t executes many functions highly relevant to f_t. Thus, new test inputs generated from such ti will likely cover b_t.

2.5 Selection of Input Bytes to Mutate

Algorithm 3: Input byte selection algorithm

1 **Function** `SelectBytes`(b_t, *input_taint*, FR)**:**

2 $bytes \leftarrow input\_taint[b_t]$

3 **foreach** $b' \in HRB(b_t)$ **do**

4 $bytes \leftarrow bytes \cup input\_taint[b']$

5 **end**

6 **return** *bytes*

FRIEND uses the following conjecture. The input bytes that affect a function g highly relevant to a target function f_t (containing a target branch b_t), will likely affect f_t and, consequently, b_t. In other words, we believe that the function relevance information can capture dependency between input bytes and a target branch in a simple and light-weight way.

In more detail, to cover b_t, FRIEND selects and mutates a subset of input bytes that affect the branch conditions of the branches *highly relevant* to a target branch b_t. By using the function relevance metric, FRIEND defines a set of branches highly relevant to a target branch b_t as following:

Definition 4. For a target branch b_t in a function f_t, we define the set of highly relevant branches $HRB(b_t)$ as follows:

$$HRB(b_t) = \{b \text{ in } g |\ g \in \{h | FR(f_t, h) > \tau\}\}$$

where τ is a user-given threshold

Algorithm 3 shows how FRIEND selects input bytes to mutate for a given target branch b_t. First, from the taint information obtained at line 21 of Algorithm 1, it selects input bytes that affect the branch condition of a target branch b_t (i.e., input bytes on which the branch condition of b_t has data-dependency) (line 2 of Algorithm 3). Then, FRIEND additionally selects input bytes that affect the branch conditions of the branches highly relevant to b_t (i.e., $HRB(b_t)$) (lines 3–5 of Algorithm 3).

2.6 Implementation

We have implemented FRIEND on top of Angora [7]. FRIEND's test input selection and input byte selection strategies are implemented in additional 1,000 lines of RUST code. To extract a list of the executed functions from a test execution, FRIEND instruments a target program using a LLVM [22] compiler pass consisting of 100 lines of C++ code. FRIEND uses a function relevance threshold τ as 0.7.

Chapter 3. Experiment Setup and Results

3.1 Experiment Setup

3.1.1 Research Questions

RQ1. Coverage Achievement Ability : To what extent does FRIEND achieve test coverage, compared to other fuzzers?

RQ2. Crash Detection Ability : To what extent does FRIEND detect crash bugs, compared to other fuzzers?

RQ3. Effect of the Test Input Selection Strategy of FRIEND: How much does the test input selection strategy affect FRIEND's coverage achievement and the number of the crashes detected, compared to FRIEND without the test input selection strategy and FRIEND with random test input selection?

RQ4. Effect of the Input Byte Selection Strategy of FRIEND: How much does the input byte selection strategy affect FRIEND's coverage achievement and the number of the crashes detected, compared to FRIEND without the input byte selection strategy and FRIEND with random input byte selection?

RQ5. Runtime Cost of Function Relevance Related Metrics: How much runtime overhead is caused by computing function relevance related metrics during the entire fuzzing process of FRIEND?

3.1.2 Target Subject Programs

I evaluated FRIEND on two different types of datasets: (1) 4 programs of LAVA-M [12], and (2) 10 real-world programs. Table 3.1 presents a type, name, version, size in terms of LoC, and a brief description of each target subject.

LAVA-M Benchmark

LAVA-M is a widely-used bug benchmark set for evaluating fuzzers. LAVA-M consists of injected bugs in four GNU coreutils programs: `base64`, `md5sum`, `uniq`, and `who`. LAVA-M authors injected 44, 57, 28, and 2136 bugs into `base64`, `md5sum`, `uniq`, and `who`, respectively. Each of the injected bug is guarded by a comparison with four-byte magic number and is triggered only if the condition is satisfied.

Real-world Programs

I have collected the latest versions (as of March 4th, 2020) of the 10 popular real-world C/C++ programs that have been extensively tested by other fuzzing tools (i.e., the latest versions of the subject programs may have only very few crash bugs left). The average size of the real-world target subjects is 72,238 LoC.

Table 3.1: Target subjects

Type	Subjects	Ver	Size (LoC)	Description
LAVA-M	base64	8.24	8937	GNU coreutils
	md5sum	8.24	9232	GNU coreutils
	uniq	8.24	9108	GNU coreutils
	who	8.24	20133	GNU coreutils
Real-world	bison	3.5.2	43797	Parser generator
Programs	bsdtar	3.4.1	87592	Archive tool
	cjpeg	2.0.4	4714	JPEG encoder
	exiv2	0.27.2	97570	EXIF parser
	nm	2.34	108933	GNU binutils
	objdump	2.34	154286	GNU binutils
	pngfix	1.6.37	8558	PNG utility
	re2	2020-03-03	50947	Regular expr. engine
	readelf	2.34	60559	GNU binutils
	size	2.34	105428	GNU binutils

3.1.3 Techniques to Compare

To answer RQ1 and RQ2, I have applied FRIEND and the following state-of-the-art fuzzers publicly available to the 10 real-world target subjects. I have compared the coverage achievement ability and crash detection ability of the fuzzers:

- *AFLFast [5]*: it is an extension of AFL [37]. AFLFast extends AFL's power schedule to assign more time to low-frequency paths (i.e., execution paths that explore interesting behaviors).
- *Angora [7]*: it is one of the state-of-the-art fuzzers which utilizes dynamic taint analysis (DTA) to select test input bytes to mutate.
- *FairFuzz [25]*: it is also an extension of AFL that searches for interesting input bytes that are worth to mutate to cover rarely executed branches.
- *RedQueen [2]*: it is one of the state-of-the-art virtual machine-based fuzzer for binary programs. Based on the observed relation between input bytes and program states, it can generate inputs that pass sanity checks efficiently.

I also compare FRIEND with the aforementioned four fuzzers and the following three fuzzers on LAVA-M to evaluate crash bug detection ability. Since the tools of the following three fuzzers are not publicly available, I used the crash bug detection results of the six fuzzers (except FairFuzz whose paper does not report its LAVA-M experiment results) reported in the corresponding papers (these papers do not report coverage result).

- *FUZZER [12]*: it is a coverage-guided fuzzer used by the LAVA-M authors.
- *SES [12]*: it is a symbolic execution tool used by the LAVA-M authors.
- *Steelix [27]*: it fuzzes binary programs. Steelix utilizes both coverage information and comparison progress information to guide test input generation.

Variants of FRIEND

To answer RQ3 (i.e., to evaluate the effect of the test input selection strategy (Section 2.4)), I applied FRIEND and its following variants to the 10 real-world target subjects.

- FRIEND $^{-IS}$: a variant of FRIEND that does not use test input selection strategy; it selects the test input that reaches the branch condition of the target branch and has the shortest execution time.
- FRIEND ^{RIS} : a variant of FRIEND that uses a *random* test input selection instead of the function relevance based one; it randomly selects a test input that reaches the branch condition of the target branch.

Similarly, to answer RQ4 (i.e., to evaluate the effect of the input byte selection strategy (Section 2.5)), I applied FRIEND and its following variants to the 10 real-world target subjects.

- FRIEND $^{-BS}$: a variant of FRIEND that does not use dynamic function relevance-based input byte selection strategy (i.e., Lines 3–5 in Algorithm 3 are removed)
- FRIEND ^{RBS}: a variant of FRIEND that uses a *random* input byte selection instead of the function relevance based one; it randomly selects the same amount of input bytes as FRIEND selects.

3.1.4 Fuzzing Setup

Timeout Setup

For LAVA-M benchmark, I ran FRIEND and FairFuzz for five hours with one CPU core, which follows the fuzzing setup for LAVA-M benchmark in the literatures [12, 7, 2] (for the other fuzzers, I used the results reported in the corresponding papers). For the real-world subjects, I ran AFLFast, Angora, FairFuzz, RedQueen, FRIEND, and the variants of FRIEND for 24 hours, which follows the guideline on evaluating fuzzers proposed by Klees et al. [21].

Testbed Setup

All the experiments were performed on our own cluster in which each node is equipped with Intel quad-core i5-9600K (3.7 Ghz) and 16GB RAM, running Ubuntu 16.04 64 bit version.

3.1.5 Measurement

Coverage Achievement

To answer RQ1, RQ3, and RQ4, for coverage measurement, I count the number of the unique execution paths reported by the fuzzing tools and the number of the covered lines and branches reported by `gcov`. Note that all fuzzers applied to measure the coverage of the real-world programs (i.e., AFLFast, Angora, FairFuzz, RedQueen, and FRIEND) share the same mechanism to probe and count a number of the unique execution paths (see [33, 21] for detail). To reduce the random variance on the experiment, I repeated the experiments 10 times and report the average numbers of the covered unique execution paths, lines, and branches of the 10 experiment results.

Crash Bug Detection

To answer RQ2, RQ3, and RQ4, I report the number of the crash bugs detected by the applied fuzzing techniques. Similarly to other fuzzing papers [4, 31], I count the number of the unique stack traces generated from the crashes as the number of crash bugs detected. To reduce the random variance on the experiment, I repeated the experiments 10 times and report the number of crash bugs detected by at least one of the 10 experiments.

3.1.6 Threats to Validity

A threat to external validity is the representativeness of our target subjects. I expect that this threat is limited since I choose the target programs widely used ones by many fuzzing researchers. LAVA-M is a de facto standard benchmark for evaluating fuzzers, and I chose real-world subjects that were used in other fuzzing studies in literature for fuzzer evaluation.

A threat to internal validity is possible bugs in the implementation of FRIEND. To control this threat, I have tested our implementation extensively.

Table 3.2: The numbers of the unique paths, lines, and branches covered by the fuzzers

	Targets	AFLFast	Angora	FairFuzz	Red-Queen	FRIEND
#unique paths covered	bison	4328	12784	3602	2305	**16660**
	bsdtar	4396	10953	4375	2082	**16531**
	cjpeg	3655	3871	3124	**5510**	5117
	exiv2	8584	3940	5961	1758	**9181**
	nm	2454	21767	2345	1979	**28203**
	objdump	6600	66672	5318	5097	**67775**
	pngfix	1138	3221	863	1076	**4318**
	re2	4741	26312	4669	7712	**35096**
	readelf	11090	16634	12896	5991	**18898**
	size	3174	22615	2790	5184	**35651**
	Average	5016.0	18876.9	4594.3	3869.4	**23743.1**
#covered lines	bison	**7541**	7131	3117	6182	7436
	bsdtar	6952	7648	7002	6584	**8078**
	cjpeg	**2044**	1901	1956	1987	1961
	exiv2	7484	**8259**	7258	7406	8252
	nm	4569	5694	4614	4518	**6018**
	objdump	8230	**11613**	9016	8488	10806
	pngfix	971	**1066**	960	1031	1024
	re2	8502	9322	8716	9514	**9611**
	readelf	10543	11307	11310	11034	**11874**
	size	4526	5560	4880	5045	**6131**
	Average	6136.2	6950.1	5882.9	6178.9	**7119.0**
#covered branches	bison	**4668**	4411	1723	3781	4471
	bsdtar	3631	3896	3672	3374	**4148**
	cjpeg	**2353**	2137	2266	2241	2201
	exiv2	6021	6710	5806	6067	**6958**
	nm	2580	3090	2647	2579	**3270**
	objdump	5154	**6454**	5268	5191	6354
	pngfix	504	531	492	**535**	533
	re2	4332	4791	4425	5033	**5042**
	readelf	7659	7697	8133	**8142**	8106
	size	2764	3206	2846	3004	**3484**
	Average	3966.6	4292.3	3727.8	3994.7	**4456.5**

3.2 Results

This section shows the experiment results to answer the five research questions.

3.2.1 RQ1:Coverage Achievement Ability

The experiment results show that FRIEND covers more unique execution paths, more lines, and more branches than the other fuzzers, on average over the 10 real-world target subjects. Table 3.2 shows the number of the unique execution paths (the 2nd to 12th rows), the lines (the 13th to 23rd rows), and the branches (the 24th to the last rows) covered by the applied fuzzing techniques for the 10 real-world target subjects. Bold cell represents the best coverage achieved among the fuzzers applied. For example, for `bsdtar`, FRIEND covers 16,531 unique paths while the second best fuzzer (i.e., Angora) covers only 10,953 paths (the third raw of the table). For another example, for `size`, FRIEND covers 6,131 source code lines while the second best fuzzer (i.e., Angora) covers only 5,560 lines (the 22nd row).

On average, FRIEND achieved the highest unique path coverage, line coverage, and branch coverage among the applied five fuzzers. FRIEND covers 373.3%, 25.8%, 416.8% and 513.6% more unique execution paths than AFLFast, Angora, FairFuzz, and RedQueen, respectively on average; it achieves the highest path coverage for the nine out of the 10 subjects (except `cjpeg`). Similarly, FRIEND covers 2.4% (Angora) to 21.0% (FairFuzz) and 3.9% (Angora) to 19.6% (FairFuzz) more lines and branches than the other fuzzers, respectively on average.

Figure 3.1 shows the trend of the unique path coverage achieved on the real-world subjects by the fuzzers for 24 hours. Note that, for most subjects, FRIEND consistently achieves higher path coverage with huge difference. Also note that the performance gap between FRIEND and the other fuzzers is getting bigger as time goes on because AFLFast, FairFuzz, and RedQueen reach a saturation point after few hours on many subjects, but the coverage of FRIEND keeps increasing.

Thus, as I have seen, I can confirm that the function relevance-based fuzzing strategies of FRIEND are effective to improve test coverage significantly.

I guess that the path coverage of RedQueen is poor because RedQueen is much slower than the other fuzzers in terms of test generation speed. This is because RedQueen is a virtual-machine based fuzzer and it frequently performs heavy kernel-level operations such as virtual machine controls to instrument a target program in a kernel level. For example of `exiv2`, the test input generation speed of FRIEND is 3.5 times faster than RedQueen (i.e.,. RedQueen generated 576 test inputs per second while FRIEND did 1995 ones per second).

3.2.2 RQ2: Crash Detection Ability

The experiment results show that FRIEND successfully detects injected bugs in LAVA-M and real-world unknown crashes. Table 3.3 shows the number of LAVA-M injected bugs detected by AFLFast, Angora, FairFuzz, RedQueen, and FRIEND. For `base64` and `uniq`, Angora, FRIEND, and RedQueen detected the same number of the bugs. For `md5sum`, RedQueen detected 61 while FRIEND did 57. For `who`, RedQueen detected 2462 while FRIEND did 2402 (i.e., FRIEND detected 2.4% less crashes than RedQueen).

For the 10 real-world subjects, however, FRIEND detected at least twice more crash bugs than the other fuzzers, which is a more meaningful indicator for crash bug detection ability for real-world application. Table 3.4 shows the number of the real-world crashes detected by AFLFast, Angora, FairFuzz,

Table 3.3: The number of the LAVA-M injected bugs detected by the fuzzers

Targets	#listed bugs	#detected bugs							
		AFL-Fast	Angora	Fair-Fuzz	FUZZER	RED-QUEEN	SES	Steelix	FRIEND
base64	28	9	29	0	7	29	0	7	29
uniq	44	0	48	0	7	48	9	43	48
md5sum	57	0	57	47	2	61	0	28	57
who	2136	1	1541	0	0	2462	18	194	2402
Total	2265	10	1675	47	16	2600	27	272	2536

Table 3.4: The number of the real-world crashes detected by the fuzzers

Targets	AFLFast	Angora	FairFuzz	RED-QUEEN	FRIEND
bison	**5**	**5**	1	0	**5**
bsdtar	0	0	0	0	0
cjpeg	0	0	0	0	0
exiv2	0	0	0	0	0
nm	2	5	0	1	**6**
objdump	0	0	0	2	**3**
pngfix	0	0	0	0	0
re2	0	0	0	0	0
readelf	0	0	0	0	**2**
size	1	1	0	4	**5**
Total	8	11	1	7	**21**

RedQueen, and FRIEND. For example, for `readelf`, FRIEND detected two crashes while the other fuzzers detected none (see the 10th row of Table 3.4). FRIEND detected 21 real-world unknown crashes from the five target subjects (`bison`, `nm`, `objdump`, `readelf`, and `size`). Compared to the second best fuzzer Angora, FRIEND detected almost twice more crashes (11 vs 21). For all of these five subjects, FRIEND detected all the bugs detected by the other four fuzzers (AFLFast, Angora, FairFuzz, and RedQueen).

3.2.3 RQ3: Effect of the Test Input Selection Strategy of FRIEND

The experiment results show that the test input selection strategy of FRIEND is effective to increase coverage achievement ability and bug detection ability. Table 3.5 shows the numbers of covered unique paths, lines, branches, and detected crashes by FRIEND $^{-IS}$, FRIEND ^{RIS}, and FRIEND. FRIEND $^{-IS}$ and FRIEND ^{RIS} cover 9.5% and 11.0% less unique execution paths than FRIEND, respectively on average. Similarly, FRIEND $^{-IS}$ and FRIEND ^{RIS} cover 1.7% and 0.9% less lines and 2.5% and 1.0% less branches than FRIEND, respectively on average. Also, FRIEND $^{-IS}$ and FRIEND ^{RIS} detect only eight crashes while FRIEND does 21. Thus, we can conclude that the dynamic function relevance-based

Table 3.5: The numbers of the covered unique paths, covered lines, covered branches, and detected crashes by FRIEND $^{-IS}$, FRIEND ^{RIS}, and FRIEND

Targets	#unique paths covered			#covered lines		
	FRIEND^{-IS}	FRIEND^{RIS}	FRIEND	FRIEND^{-IS}	FRIEND^{RIS}	FRIEND
bison	**19363**	13885	16660	**7465**	7140	7436
bsdtar	12915	10126	**16531**	7906	7946	**8078**
cjpeg	4034	4879	**5117**	1876	1958	**1961**
exiv2	4027	7891	**9181**	8091	**8391**	8252
nm	25702	27268	**28203**	5678	5759	**6018**
objdump	**68132**	64599	67775	11376	**11495**	10806
pngfix	3842	3877	**4318**	1019	1022	**1024**
re2	34710	33521	**35096**	**9748**	9623	9611
readelf	**18901**	18616	18898	11495	11528	**11874**
size	25141	29276	**35651**	5563	5985	**6399**
Average	21676.7	21393.8	**23743.1**	7021.7	7084.7	**7145.8**

Targets	#covered branches			#detected crashes		
	FRIEND^{-IS}	FRIEND^{RIS}	FRIEND	FRIEND^{-IS}	FRIEND^{RIS}	FRIEND
bison	**4514**	4447	4471	**5**	**5**	**5**
bsdtar	4084	4063	**4148**	0	0	0
cjpeg	2131	2197	**2201**	0	0	0
exiv2	6662	**7054**	6958	0	0	0
nm	3083	3145	**3270**	2	2	**6**
objdump	6367	**6387**	6354	0	1	**3**
pngfix	529	**533**	**533**	0	0	0
re2	5031	5017	**5042**	0	0	0
readelf	7863	7926	**8106**	0	0	**2**
size	3211	3379	**3484**	1	0	**5**
Average	4347.5	4414.8	**4456.5**	8	8	**21**

test input selection strategy is effective to improve the coverage achievement ability and bug detection ability of FRIEND.

3.2.4 RQ4. Effect of the Input Byte Extension Strategy of FRIEND

The experiment results show that the input byte selection strategy of FRIEND is effective to increase coverage achievement ability and bug detection ability. Table 3.6 shows the numbers of covered unique paths, lines, branches, and detected crashes by FRIEND $^{-BS}$, FRIEND ^{RBS}, and FRIEND. FRIEND $^{-BS}$ and FRIEND ^{RBS} cover 18.1% and 15.6% less unique execution paths than FRIEND, respectively on average. Similarly, FRIEND $^{-BS}$ covers 2.4% less lines. FRIEND $^{-BS}$ and FRIEND ^{RBS} cover 2.5% and 1.0% less branches than FRIEND, respectively on average. Also, FRIEND $^{-BS}$ and FRIEND ^{RBS}

Table 3.6: The numbers of the covered unique paths, covered lines, covered branches, and detected crashes by FRIEND $^{-BS}$, FRIEND ^{RBS}, and FRIEND

Targets	#unique paths covered			#covered lines		
	FRIEND^{-BS}	FRIEND^{RBS}	FRIEND	FRIEND^{-BS}	FRIEND^{RBS}	FRIEND
bison	18241	**19219**	16660	7435	**7465**	7436
bsdtar	11744	9129	**16531**	7613	7995	**8078**
cjpeg	4269	3763	**5117**	1854	1986	**1961**
exiv2	5569	5211	**9181**	8205	**8365**	8252
nm	23211	21002	**28203**	5971	**6552**	6018
objdump	61710	58700	**67775**	10308	**11143**	10806
pngfix	3598	3267	**4318**	1022	**1061**	1024
re2	26710	27585	**35096**	9419	9376	**9611**
readelf	16429	18713	**18898**	11348	11754	**11874**
size	22929	33881	**35651**	**6579**	6351	6399
Average	19441.0	20047.0	**23743.1**	6975.4	**7204.8**	7145.8

Targets	#covered branches			#detected crashes		
	FRIEND^{-BS}	FRIEND^{RBS}	FRIEND	FRIEND^{-BS}	FRIEND^{RBS}	FRIEND
bison	4484	**4514**	4471	**5**	**5**	**5**
bsdtar	**4510**	4090	4148	0	0	0
cjpeg	2154	2093	**2201**	0	0	0
exiv2	6801	**6966**	6958	0	0	0
nm	3208	**3403**	3270	2	4	**6**
objdump	6129	6290	**6354**	0	0	**3**
pngfix	531	526	**533**	0	0	0
re2	4890	4887	**5042**	0	0	0
readelf	7633	8006	**8106**	0	0	**2**
size	**3603**	3544	3484	0	1	**5**
Average	4394.3	4431.9	**4456.5**	7	10	**21**

detect only seven and 10 crashes respectively while FRIEND does 21.

3.2.5 RQ5. Runtime Cost of Function Relevance Related Metrics

The experiment results show that the runtime cost to calculate the dynamic function relevance related metrics is negligible. Table 3.7 shows the execution time taken to compute the dynamic function relevance related metrics (in minutes) and the ratio of the computation time to the total fuzzing time (24 hours). For example, for `readelf` (see the fifth column of the fifth row), it takes only 2.1 minutes to compute the dynamic function relevance metric, which is only 0.1% of the total fuzzing time.

On average over the 10 real-world subjects, the total time to compute the function relevance related metrics is taken 14 minutes. `re2` consumes the longest metric computation time (49.4 minutes), but it

Table 3.7: Execution time (in minutes) taken to compute the dynamic function relevance metric and the ratio of the execution time to the whole fuzzing time (i.e., 24 hours)

Targets	Time (mins)	Ratio(%)	Targets	Time (mins)	Ratio(%)
bison	4.6	0.3%	objdump	11.4	0.8%
bsdtar	2.2	0.1%	pngfix	32.7	2.3%
cjpeg	6.4	0.4%	re2	49.4	3.4%
exiv2	12.2	0.8%	readelf	2.1	0.1%
nm	9.5	0.7%	size	9.2	0.6%
	Average			14.0	1.0%

is still only 3.4% of the total fuzzing time. FRIEND can compute the metrics fast because it only needs to track which function is executed during the execution of a target program (this can be implemented very efficiently by using a bit-vector whose bit is set to one if a corresponding function is invoked).

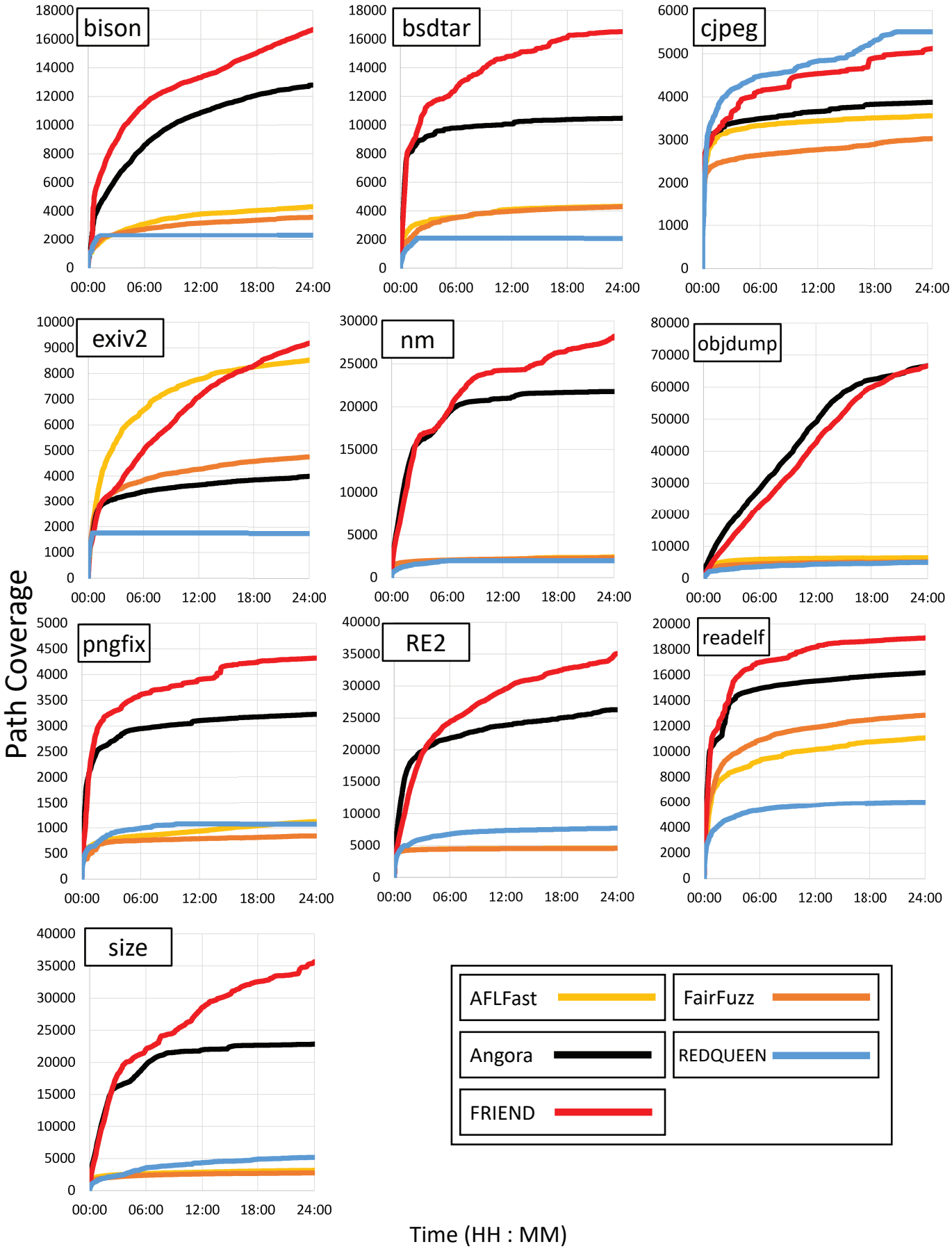

Figure 3.1: The average unique path coverage achieved by the fuzzers for 10 runs, 24 hours.

Chapter 4. Related Works

Coverage-guided fuzzing repeats the following three main steps within a given time budget:

1. to select/schedule a *test input* to mutate,
2. to select which *input bytes* to mutate in the selected test input, and
3. to put *proper values* for selected input bytes to generate new test inputs

I described related fuzzing works with respect to test input selection strategies, input byte selection strategies, and dictionary-based mutation strategies which is a common value selecting strategy.

4.1 Fuzzing Techniques to Select/Prioritize Test Inputs

Test input schedule strategy is also named as test input selection or test input prioritization. Many fuzzers prioritize test inputs by scoring each test input in terms of its byte size, execution speed, path frequency and so on.

AFL [37] favors test inputs with new path coverage, small byte size, and less execution time. AFLFast [5] favors test inputs that execute rarely executed paths. FairFuzz [25] and Vuzzer [31] favors test inputs which execute rarely executed branches and which execute basic blocks located in deep control-structure, respectively. CollAFL [15] favors test inputs whose execution paths have many uncovered neighbor branches. Ankou [28] defines a distance between two different execution paths and it scores each test input according to its execution path's "uniqueness" which is measured using the distances to other paths.

TortoiseFuzz [34] favors test inputs which execute many functions, loops, and basic blocks that have many memory access operators. SAVIOR [9] statically label suspicious basic blocks which contain (or which can reach) operators that could lead to undefined behaviors, and it scores each test input in terms of a number of the suspicious basic blocks visited by the test input.

Angora [7] selects only test inputs which execute uncovered branches without prioritization among the selected test inputs.

Unlike the above fuzzers that try to achieve high coverage or to find many crashes, some fuzzers select test inputs to satisfy their own objectives such as detecting memory bugs and/or performance bugs. For example, MemLock [35] focuses to detect memory consumption bugs by selecting and mutating test inputs that consume huge memory. PerfFuzz [24] targets to generate test inputs that reveal performance issues by selecting and mutating the test inputs with high total execution path length. AFLGo [4] aims to generate test inputs which execute given target locations of program; it defines a distance between a test input and target locations using static analysis, and favors test inputs with low distance.

4.2 Fuzzing Techniques to Select Input Bytes to Mutate

Since the size of execution search space for fuzzing increases exponentially to the length of the input bytes to mutate, fuzzers identify few important input bytes using various techniques and mutate only those important bytes.

Dynamic Taint Analysis (DTA) BuzzFuzz [16], Dowser [18], Vuzzer [31], Angora [7] and Matryoska [8] use DTA to determine which bytes to mutate for covering a target branch. Angora selects and mutates only input bytes on which a target branch has data dependency. Matryoska uses both DTA and a heuristic that it mutates an input byte in_j such that mutating in_j does not change a previous execution path that reaches a target branch condition.

Dynamic Inference on Input Bytes Some fuzzers avoid DTA due to DTA's high runtime overhead, and they select input bytes to mutate by analyzing dynamic execution information. FairFuzz [25], Profuzzer [36], and GreyOne [14] try byte-level mutation to discover "interesting " bytes that show interesting symptoms. If a test input obtained by mutating an input byte executes a rarely-execute branch, FairFuzz marks the input byte as an interesting one to mutate. Profuzzer tries to identify a type (e.g., int, long, structure) of a byte segment in input bytes by trying byte-level mutation on every input byte. Once a type of input bytes is identified, it applies mutation strategies adequate to the type of the input. GreyOne records each program variable's value changes to infer which input bytes have dependency with the program variables in uncovered branches on the pilot fuzzing stage. Although FRIEND does not explicitly make inference on inputs bytes like these fuzzers, it still obtains valuable information on input bytes through function relevance information.

4.3 Dictionary-based Mutation in Fuzzing Techniques

Although select proper bytes can significantly increase probability to generate new meaningful test inputs, new value to put in mutated bytes is also important. Dictionary-based mutation was developed for effective fuzzing for simply structured input files (to complexly-structured input files, grammar-based fuzzing [38, 39, 40] are applied). It puts word values in its dictionary into input bytes to generate new test inputs that contains meaningful values.

The dictionary consists of tokens provided by users or automatically extracted from target programs' source code and/or documents. Dictionary-based mutation adds or deletes a token and mutates one token into another to effectively generate test inputs that satisfy the input constraints of the target programs. To guide fuzzing an input file, AFL [37] provides an API to use either a user-provided dictionary or an automatically extracted one. Superion [39] is also an example of dictionary-based mutation fuzzing. It proposed an improvement on AFL's dictionary-based mutation to align with their grammar-aware fuzzing.

Chapter 5. Concluding Remark

This dissertation presented FRIEND which outperforms the state-of-the-art fuzzers in terms of test coverage and crash bug detection. A core idea of FRIEND is to utilize "dynamic function relevance" to select test inputs and input bytes to mutate, which is a salient method to improve test coverage and crash bug detection ability cost-effectively. The evaluation on the four LAVA-M benchmark programs and 10 popular real-world programs showed that FRIEND achieves significantly higher test coverage and detected more crash bugs than the state-of-the-art fuzzers (AFLFast, Angora, FairFuzz, and RedQueen).

As future work, I will make a more general version of the test input and input byte selection strategies based on dynamic function relevance , which can be applied to any of the coverage-guided fuzzers. Also, I will refine the function relevance metric by utilizing various dynamic and static code features with machine learning techniques. Finally, I will apply FRIEND to multi-million LoC programs such as Apache and PHP to show that FRIEND can detect crash bugs in an extremely large program.

Bibliography

[1] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software", *IEEE Transactions on Software Engineering*, volume 30, number 8, pp. 491-506, 2004.

[2] S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence", *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques", *ACM Comput. Surv.*, volume 51, number 3, articleno 50, pp. 39, May, 2018.

[4] M. B"ohme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing", *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329-2344, 2017.

[5] M. B"ohme, V. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain", *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, pp. 1032-1043, 2016.

[6] C. Cadar, and K. sen, "Symbolic Execution for Software Testing: Three Decades Later", *Commun. ACM*, volume 56, number 2, pp. 82-90, Feb., 2013.

[7] P. Chen, and H. Chen, "Angora: Efficient Fuzzing by Principled Search", *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711-725, 2018.

[8] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing Deeply Nested Branches", *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 499-513, 2019.

[9] Y. Chen, P. Li, J. Xu, S. guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards Bug-Driven Hybrid Testing", *2020 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, pp. 2, 2020.

[10] S. Chidamber, and C. Kemerer, "Towards a Metrics Suite for Object Oriented Design", *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1991.

[11] S. Chidamber, and C. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*. volume 20, number 6, pp. 476-493, 1994.

[12] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition", *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 110-121, 2016.

[13] C. Evans, M. Moore, and T. Ormandy, "Fuzzing at Scale", *https://security.googleblog.com/2011/08/fuzzing-at-scale.html*.

[14] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing", *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020.

[15] S. Gan, C. Zhang, Z. Pei, and Z. Chen, "2018 IEEE Symposium on Security and Privacy (SP)", *CollAFL: Path Sensitive Fuzzing*, volume 0, pp. 660-677, 2018.

[16] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing", *2009 IEEE 31st International Conference on Software Engineering*, pp. 474-484, 2009.

[17] G. A Hall, W. Tao, J. Munson, "Measurement and validation of module coupling attributes", *Software Quality Journal*, volume 13, number 3, pp. 281-296, 2005.

[18] I. Haller, A. Slowinkska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations", *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 49-64, 2013.

[19] Y. Kim, Y. Choi, and M. Kim, "Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering", *International Conference on Software Engineering (ICSE)*, pp. 315-326, 2018.

[20] J. C. King, "Symbolic Execution and Program Testing", *Commun. ACM*, volume 19, number 7, 1976.

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

[22] C. Lattner, V. Adve, "LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation", *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[23] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring Coupling and Cohesion of Object-Oriented Programs based on Information Flow", *International Conference on Software Quality (ICSQ)*, 1995.

[24] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs", *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pp. 254-265, 2018.

[25] C. Lemieux, and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage", *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, pp. 475-485, 2018.

[26] W. Li and S. Henry, "Object-oriented metrics that predict maintainability", *Journal of Systems and Software*, pp. 111-122, 1993.

[27] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing", *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pp. 627-637, 2017.

[28] V. JM Manès, S. Kim, S. Cha, "Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference", *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1024-1036, 2020.

[29] V. JM Manès, H. Han, C. Han, S. Cha, M. Egele, E. Schwartz, and M. Woo "The art, science, and engineering of fuzzing: A survey", *IEEE Transactions on Software Engineering*, 2019.

[30] M. Moroz, and K. Serebryany, "Guided in-process fuzzing of Chrome components", *https://security.googleblog.com/2016/08/guided-in-process-fuzzingof-chrome.html*, 2016.

[31] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing", *NDSS*, volume 17, pp. 1-14, 2017.

[32] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C", *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263-272, 2005.

[33] 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", *NDSS*, volume 16, pp. 1-16, 2016.

[34] Y. Wang, X. Jia,. Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization", *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[35] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "MemLock: Memory Usage Guided Fuzzing", *2020 IEEE/ACM 42nd International Conference on Software Engineering*, 2020.

[36] 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", *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 769-786, 2019.

[37] M. Zalewski, American fuzzy lop (AFL) fuzzer. *http://lcamtuf.coredump.cx/afl/*.

[38] T. Pham, M. Boehme, A. Santosa, A. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing", *IEEE Transactions on Software Engineering*, 2019.

[39] J. Wang, B. chen, L. Wei, Y. Liu, "Superion: Grammar-aware greybox fuzzing", *Proceedings of the 41st International Conference on Software Engineering*, pp. 724-735, 2019.

[40] C. Holler, K. Herzig, and A. Zeller "Fuzzing with code fragments", *21st USENIX Security Symposium (USENIX Security 12)*, pp. 445-458, 2012.

Acknowledgments in Korean

길고도 짧았던 2년이라는 시간 동안 석사 생활을 무사히 마칠 수 있게끔 도와준 모든 분들에게 감사를 드립니다. 항상 믿고 지켜봐주시는 부모님과 가족들에게 감사드립니다. 6년이 넘는 시간 동안 먼 타지에서도 계속해서 공부하고 연구를 지속할 수 있었던 것은 항상 가족분들 덕분입니다.

4년여의 시간동안 한결같이 지도를 해주신 김문주 교수님과 김윤호 교수님께 감사드립니다. 제가 처음 연구에 발을 들였을 때, 많이 부족했던 제가 연구 활동을 할 수 있게 된건 교수님의 사랑깊은 지도가 있었기에 가능했습니다. 주신 사랑 제가 조금이나마 보답드릴 수 있도록 노력하겠습니다.

제가 연구실에 있던 동안 연구실을 다녀가신 임현수, 김현우, 박건우, Loc Duy Phan, 김진솔 선배분들, 그리고 같이 연구를 진행하며 동고동락한 Zidong Yang, Robert Sebastian Herlim, Irfan Ariq 동료 분들, 같이 연구실 생활을 한 이동희 연구원님, 김준삼 박사님, 지준왕 선교사님, 김보경 선생님, 그리고 처음 산학과제를 진행했을 때 도움을 주신 이동주 책임님과 백준기 연구원님께도 감사들 전합니다. 이 글에는 못 담았지만, 학사시절을 포함하여 많은 도움을 받았습니다. 감사드립니다.

이 글을 읽는 모든 분이 이글로써 도움이 되었기를, 행복하시기를 바랍니다.

Curriculum Vitae in Korean

이　　름: 이 아 청

학　　력

2013. 3. – 2015. 2.　한성과학고등학교 (2년 졸업)

2015. 3. – 2019. 8.　한국과학기술원 전산학부 (학사)

2019. 9. – 2021. 8.　한국과학기술원 전산학부 (석사)

경　　력

2019. 8. 20.　삼성전자DS-KAIST 전략산학 4차년도 최종교류회 우수 포스터상

2019. 12. 18-20　Korea Software Congress (KSC) 우수 논문상

연 구 업 적

1. 이아청, 김현우, 김윤호, 김문주, Bitfield 심볼릭 지원을 통한 Concolic 테스팅 효과 향상, Korea Computer Congress (KCC), Jun 20-22, 2018
2. 이아청, 김윤호, 김문주, 함수 관련도를 이용한 퍼징의 커버리지 향상, Korea Software Congress (KSC), Dec 18-20, 2019
3. 이아청, 김윤호, 김문주, 동적 함수 관련도를 이용한 퍼징 커버리지 향상 기법, Journal of KIISE, Vol.48, Num. 4, Apr 2021