석 사 학 위 논 문
Master's Thesis

C++ 프로그램의 유닛 테스트 자동 생성 기술 분석 및 개선

On Understanding and Improving Automated Unit-level Test Generation for C++ Programs

2022

헤림 로버트 세바스티안 (Herlim, Robert Sebastian)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

C++ 프로그램의 유닛 테스트 자동 생성 기술 분석 및 개선

2022

헤림 로버트 세바스티안

한 국 과 학 기 술 원

전산학부

C++ 프로그램의 유닛 테스트 자동 생성 기술 분석 및 개선

헤림 로버트 세바스티안

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 12월 15일

심사위원장 김 문 주 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 백 종 문 (인)

On Understanding and Improving Automated Unit-level Test Generation for C++ Programs

Robert Sebastian Herlim

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
December 15, 2021

Approved by

Moonzoo Kim
Associate Professor of School 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 헤림로버트 세바스티안. C++ 프로그램의 유닛 테스트 자동 생성 기술 분석 및 개선. 전산학부 . 2022년. 52+iv 쪽. 지도교수: 김문주. (영문 논문)
Robert Sebastian Herlim. On Understanding and Improving Automated Unit-level Test Generation for C++ Programs. School of Computing . 2022. 52+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

C++는 그 확장성, 유연성 및 고성능을 가진 프로그래밍 언어로써 많은 응용 프로그램에 널리 사용되고 있다. 하지만, C++는 복잡한 문법 체계를 가지고 있어, 정확한 코드를 적기 위해서는 매우 어려우며, 따라서 C++ 프로그램을 자동으로 테스트하여 코드의 질을 높일 수 있는 기술이 필요하다. 하지만 템플릿 인스턴스화, 복잡한 STL 타입 등의 높은 복잡성을 가지는 C++ 코드의 분석이 어려워, 실제 C++ 프로그램을 유닛 테스팅 할 수 있는 도구는 거의 없는 실정이다.

본 논문에서는 위에서 언급된 C++의 복잡성을 해결하여 자동으로 C++ 프로그램을 유닛 테스팅 할 수 있는 도구인 CITRUS를 개발하여 제안한다. CITRUS는 주어진 C++ 프로그램 P를 분석하여, P의 다양한 함수 호출을 포함하는 테스트 드라이버를 자동으로 생성한다. 더 나아가서, 테스트 드라이버를 변이하여 더 다양한 함수 호출 시퀀스를 생성하고, `libfuzzer`를 이용하여 각 함수 호출의 인수 (argument)를 변이하여 다양한 값으로 함수를 호출 할 수 있도록 한다. CITRUS를 평가하기 위해 실제 C++ 프로그램에 적용한 결과, 95%의 명령문(statement) 커버리지 향상과, 79%의 분기 커버리지 향상을 보였다.

핵 심 낱 말 자동 테스트 생성, C++ 유닛 테스트, 랜덤 함수 호출 시퀀스 생성, 코드 돌연변이, 동적 분석.

Abstract

C++ is popular in many application domains for its extensibility, flexibility, and high performance. At the same time, however, C++ is infamous for its complex syntax and semantics. Thus, it is challenging to write correct C++ programs and the need to automatically test C++ programs has been high. Unfortunately, due to the high complexity of C++ (e.g., template instantiation, complex STL types, etc.), there are almost no automated unit testing tool publicly available for real-world C++ programs.

I have developed a new automated unit testing tool *CITRUS* that resolves the aforementioned complexity of C++ programs. After analyzing the source code of a target C++ program P, CITRUS automatically generates test driver files for P, each of which consists of various method calls of P. Then, to improve the test coverage of P, it generates more diverse test drivers by mutating the test driver code. Also, CITRUS increases the test coverage of P further by applying `libfuzzer` to alternate P's state by mutating arguments of the methods. I have demonstrated the testing effectiveness and the efficiency of CITRUS through the experiments on the real-world C++ programs, on which CITRUS achieves up to 95% statement and 79% branch coverage.

Keywords Automated test generation, C++ unit testing, random method sequence generation, code mutation, dynamic analysis.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Background

1.1.1 Research Background

Automated test case generation has become one prominent research topic in the software engineering field for the past decade [1, 2, 3, 4]. Various techniques have been proposed, ranging from the blackbox random testing [1, 5], coverage-guided greybox fuzzing [6, 7, 8], dynamic symbolic execution [9, 10], into the sophisticated AI-driven search-based software testing (SBST) [11, 2], among which each introduces its own strengths and limitations [12]. The adoption of automated test case generation has been drawing attention recently as it has been verified to be useful for practical uses, such as uncovering security vulnerabilities in modern systems [13, 14, 7, 15] and locating program faults in real-world industrial cases [16]. Moreover, a recent study has shown that automated test case generation produces test cases with higher test coverage compared to the manually-written test cases [17]. Those success stories of automated software testing strongly motivates the necessity of automated approaches for testing modern softwares in the future.

Automated testing has been applied to generate millions of inputs to explore program states in system-level executions [6, 7, 8, 18]. Unfortunately, system-level testing frequently fails to reach vulnerable statements located deep within the code due to many reasons, such as the highly-complex input specification in system-level, and so on. For example, environmental constraints may also hinder the effectiveness of test case generation in embedded systems [10]. To overcome such problem, some recent works on automated test case generation ultimately focus on generating high-coverage test cases in unit-level [1, 2, 10, 19]. However, for some highly-complex programming languages such as C++, the availability of well-performing automated testing tools in unit-level is strictly limited.

This work focuses on automated unit-level test case generation for object-oriented programs written in C++ programming language. C++ programming language is famous for its extensibility, flexibility, and high performance. Thus, it is popular in many application domains that requires significant development efforts such as database engines, operating systems, web browsers, video games, and so on. However, due to the notoriously high complexity of the syntax and semantics of C++, it is technically challenging to develop reliable C++ programs even to the current days. Moreover, although the need to automatically test C++ programs has been high, there are *almost no* automated unit testing tool publicly available for real-world C++ programs due to the high complexity of C++ language features (e.g., template instantiation, complex STL types, and so on).

1.1.2 Previous Approaches

Currently, there are only very few works on automated unit-level test case generation for C++ programs. Meanwhile, the development of automated testing tools for unit-level testing has been evolving rapidly for other programming languages, such as Java (e.g., Randoop [1] and EvoSuite [2]) and Python (e.g., Pynguin [19]). EvoSuite [2], for example, has been well-known to be the current state-of-the-art tools to generate JUnit tests *automatically* from an existing Java programs. However, due to the

syntactical differences in programming features and language characteristics, these tools can only target programs written in their specific programming language and are not applicable to target C++ programs.

The system-level approaches, such as coverage-guided fuzzing tools [6, 14, 18] and symbolic executions [9, 10], are difficult to be used (but yet still applicable) to test C++ programs in unit-level. Fuzzing tools generate and mutate input bytes to C++ target programs (without analyzing the language complexity of C++ program code). Symbolic execution approaches generate concrete values for each symbolic variables based on running constraint solvers on the collected symbolic path formulas. While such system-level approaches are still applicable for unit-level testing, the major disadvantage of using system-level approaches in unit-level testing is the high cost requirement to provide numerous stub drivers for each unit (*a.k.a.* function). Moreover, the limited testing time budget and the large search space of possible ordering of method calling sequences may also hinder the effectiveness of such system-level approaches.

To mitigate the high cost of generating stub drivers in unit-level automated testing, more recent approaches [20, 21, 22, 23] automatically generate the stub drivers at the initial stage of the testing to act as the entry point for each unit. KLOVER [20] and FSX [21] generate static drivers initially to declare the symbolic variables to be passed for each function arguments. However, such static drivers do not incorporate the possibility of varying method call sequence ordering to diversify the produced object states, which is one of the preeminent features to test in object-oriented programming language. Meanwhile, FUDGE [22] and FuzzGen [23] synthesize unit-level stub drivers of the target library APIs by referring to existing code snippets in external library consumer projects. Referring to the existing code snippets *indeed* helps FUDGE and FuzzGen to prevent API misuses on stub drivers, *but* relying onto the existence of such snippets (e.g., from only a single consumer project) may fail on producing stub drivers for uncommonly used APIs.

1.2 Thesis Statement and Contributions

1.2.1 Thesis Statement

The thesis statement for this work is written as follows:

> An automated testing tool based on method call sequence generation can generate high-coverage test cases for C++ programs that address the notoriously high complexity of C++ language features in unit-level testing context.

1.2.2 Proposed Approach

To resolve the aforementioned difficulties of testing C++ programs, I have developed a new automated unit testing tool CITRUS (**C**++ un**I**t **T**esting for **R**eliable and **U**sable **S**oftware) (Section 2.6.2 describes how CITRUS handles template instantiation and Section 2.6.3 shows how it manages complex STL types). CITRUS receives the source code of a target C++ program P and it automatically generates test driver files for P, each of which consists of various method calls of P. Then, CITRUS generates more diverse test drivers by mutating the test driver code to increase the test coverage of P. In other words, CITRUS explores diverse states of P by executing various method sequences of functions in P.

In addition, CITRUS improves the test coverage of P further by applying `libfuzzer` [24] to change P's state by mutating arguments of the methods.

I have demonstrated the testing effectiveness and the efficiency of CITRUS through the experiments on the eight real-world C++ programs (`jsonbox`, `hjson`, `tinyxml2`, `jvar`, `jsoncpp`, `json-voorhees`, `yaml-cpp`, and `re2`) ranging from 1.5 KLoC to 20 KLoC. On these target programs, CITRUS achieved up to 95% statement (on `jsoncpp`) and 79% branch (on `jsonbox`) coverage.

1.2.3 Contributions

The contributions of this work are listed as follows:

1. CITRUS generates test cases on real-world C++ programs in fully-automated manner. Compared to previous works on unit-testing C++ programs in system-level approaches (e.g., symbolic execution), CITRUS automatically generates drivers of the target program and does not require any human tester to interfere during the process.

2. Due to the high complexity of C++ language features, CITRUS is one of the very few tools that can automatically generate unit-level test cases to work on complex real-world C++ programs. In the past year, I have worked on CITRUS by putting extensive engineering efforts to accommodate the rich language features of C++, such as template classes, STL classes, inheritances, implicit constructors, and so on.

3. I have performed an in-depth evaluation on eight real-world C++ programs to evaluate the testing effectiveness and efficiency of CITRUS. In my experiment, CITRUS achieved high testing effectiveness (up to 95.4% for line coverage and 78.9% for branch coverage). Also, by continually fuzzing using `libfuzzer` after the method sequence generation stage, CITRUS produced a competitively good result in significantly less testing time budget (i.e., saving up to 14.6 hours in average).

4. I have reported a concrete case study to demonstrate how CITRUS detects crash bugs on the C++ programs, which remains as a challenging task in automated unit-level testing due to its proneness of false alarms. The case study is elaborated in Section 3.3.1.

5. CITRUS has been made as an open-source tool to further support the future development of automated unit-level test case generation for C++ programs. The source code of CITRUS is publicly available at https://github.com/swtv-kaist/CITRUS.

1.3 Structure of Thesis

The remaining chapters of this thesis are structured as follows: Chapter 2 describes the design of CITRUS's overall process on how it generates various test cases. Chapter 3 describes the experiment setup and evaluation results to demonstrate the testing effectiveness and efficiency of CITRUS. Chapter 4 elaborates some lessons learned from this work and provides several suggestions to improve CITRUS. Chapter 5 explains some related works to automated test case generation for C++ programs based on method call sequence generation. Finally, Chapter 6 summarizes the thesis with conclusion and future works.

Chapter 2. CITRUS: C++ Unit Testing for Reliable and Usable Software

2.1 Overview

CITRUS adopts random method call sequence generation to generate test suites that extensively exercise the target program in unit-level testing. A method call sequence that either: (1) contributes to the test coverage, or (2) induces crash on the target program; will be kept as interesting test cases. Then, CITRUS generates `libfuzzer` harness drivers from the non-crashing test cases to continue traversing the target program. Figure 2.1 illustrates the overview process of CITRUS as described above.

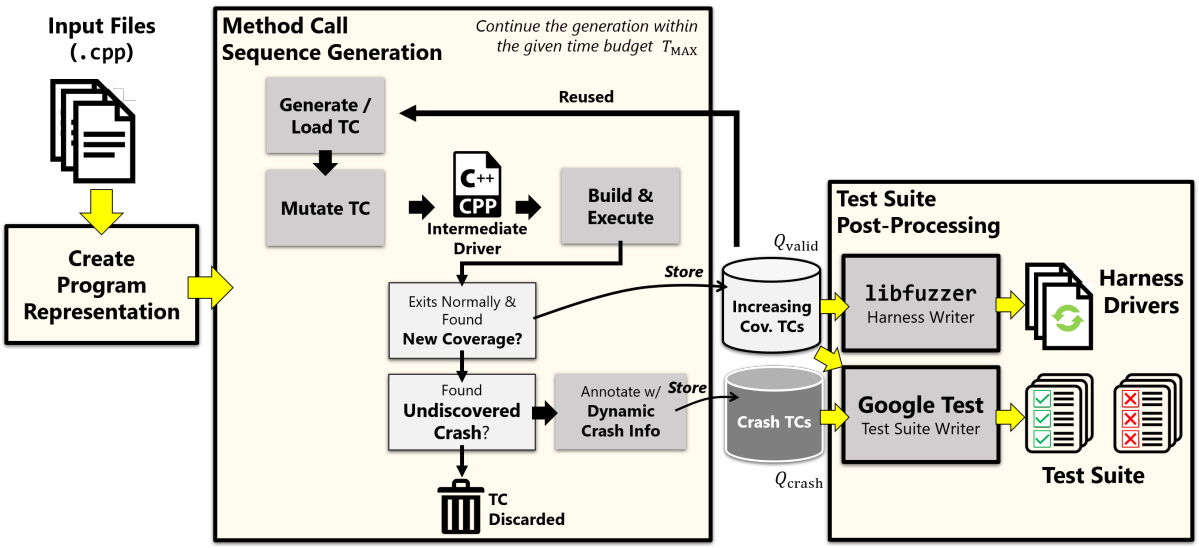

Figure 2.1: Overview of CITRUS's process

2.2 Motivating Example

This section shows *four* motivating examples to further elaborate the key challenges in automated unit-level test case generation tools for C++ programs.

Requires unit-level harness drivers. There has been several *program input*-based mutation approaches to test C++ programs in system-level testing, such as coverage-guided fuzzing [6, 18] and symbolic execution [25]. However, such approaches are difficult to be applied on unit-level testing as they require manual construction the unit-level harness drivers prior to the test input generation.

Listing 2.1 demonstrates that manual testing requires numerous harness drivers to construct diverse object states to extensively test `Printer::ToJSON` method. Some classes may have sufficiently many fields within their internal representation, such as `APIResponse` class has 100 unique *setter* methods as shown in Listing 2.1. This means, in a pairwise combinatorial testing scenario, the tester must manually write 4,950 ($=_{100}C_2$) combinations of possible pair of setter calls to apply unit-level testing using the system-level approaches mentioned above.

Listing 2.1: Motivating Example I: Requires unit-level harness drivers

```
1: class APIResponse {
2:   public:
3:     void SetField1(int arg) { ... }
4:     void SetField2(const char* str, int len) { ... }
5:     ... // and another 98 setter functions for each field
6: };
7: class Printer {
8:   public: void Print(const APIResponse& response) { ... }
9: };
```

Listing 2.2: Motivating Example II: Object construction problem (OCP)

```
 1: class Shape {
 2:   public: virtual void Draw() = 0; // pure virtual function
 3: };
 4: class Circle : public Shape {
 5:   public: void Draw() { ... }
 6: };
 7:
 8: enum class ColorType { WHITE, BLACK, YELLOW };
 9: class AlphaColor {
10:   private: AlphaColor(ColorType ctype, double alpha) { ... }
11: };
12: AlphaColor BaseAlphaColor() { ... }
12: void PrintWithColor(const Shape &shape, const AlphaColor &color);
```

Object creation problem (OCP). The next challenge after writing harness drivers is the object creation problem (OCP). The OCP is defined as how do we construct valid method call sequences to obtain an instance of a particular class. Some classes may have non-trivial ways to construct, such as implicit constructors and static factory methods (see Section 2.4.1 for details of these two concepts).

Listing 2.2 demonstrates a concrete example of OCP on generating valid sequence of method calls to test `PrintWithColor` (L13). The `PrintWithColor` method takes two `const` *rvalue* reference arguments: `Shape` and `AlphaColor`. However, instances construction of `Shape` and `AlphaColor` are non-trivial as described below.

1. First, a direct instantiation of `Shape` class forms an invalid sequence because `Shape` is an *abstract base class* (notice the `Shape::Draw` at line 2 is a *pure virtual* function). To satisfy the `Shape` argument requirement, class instances from the `Circle` class (which is a subtype of `Shape`) should be used. Note that the construction of a `Circle` instance involves an implicit default constructor (i.e., "`Circle()`") deduction, which might be non-trivial for automated tools.

2. Second, a direct invocation to `AlphaColor`'s constructor also produces uncompilable code (because `AlphaColor`'s constructor was marked private). To construct `AlphaColor`, automated tools must find another existing API that returns an instance of `AlphaColor` (e.g., `BaseAlphaColor()` at L12).

Listing 2.3: Motivating Example III: Testing C++ template classes

```
1: template<typename U, typename V>
2: class Pair {
3:   public:
4:     Pair(U first, V second);
5:     bool Equal(const Pair<U, V> &other);
6: };
```

Listing 2.4: Motivating Example IV: Integration with STL classes

```
1: vector<vector<int>> Transpose(const vector<vector<int>> &matrix);
2: void DrawAllShapes(const array<Shape*, 100> &shapes);
```

Testing C++ template classes. Testing template classes in C++ introduces another difficult challenge in C++ automated unit-level testing due to several reasons, such as:

1. It is almost impossible to instantiate template classes with all possible types in the target program.
2. Template classes introduce *unbounded* type variables (e.g., `T`, `U`, `V`) which are hard to determine for automated tools. In other words, automated tools must be able to cleverly infer the corresponding concrete types for each unbounded type variables while generating method call sequences.

At Listing 2.3, to invoke `Pair::Equal` method (L5) on a `Pair<double, int>` instance, automated tools must be aware that type variables `U` and `V` correspond to the types `double` and `int` respectively. Note that the `Equal`'s signature only *hints* an unbounded `Pair<U, V>` type as argument, and automated tools must infer the corresponding type bindings correctly. Then, the tools must generate method call sequences to construct an instance of `Pair<double, int>` (i.e., using the constructor at line 4) for a successful `Pair::Equal` method invocation.

Integration with STL classes. C++ provides Standard Template Library (STL) which is already being widely-used by most C++ programs. This implies that C++ testing tool must also accommodate each class provided in the STL used by the target program. For example, as shown in Listing 2.4, automated tools must be aware of the construction process of a multidimensional vector (i.e., `vector<vector<int>>`) to invoke `Transpose` method at line 1. Similarly, to invoke `DrawAllShapes` at line 2, automated tools must be able to construct the *fixed-sized* STL's `array` (i.e., with exactly 100 elements), because passing an `array` instance with mismatched size (e.g., `array<Shape*, 50>`) leads into an invalid call sequence (i.e., raises compilation error).

In summary, all of those four key challenges happen frequently while testing C++ programs in unit-level. CITRUS has been implemented carefully by putting heavy engineering efforts to mitigate all C++ challenges mentioned above. The following subsections will further describe how CITRUS works, starting by the definition of a CITRUS test case.

2.3 Test Cases Generated by CITRUS

2.3.1 CITRUS Test Case Definition

A test case generated by CITRUS has the following characteristics. A CITRUS test case tc is defined as a sequence of method invocation statements and all *supporting* statements to construct arguments of the method calls. Each CITRUS test case mostly triggers a particular unique behavior of the target program, where such interesting behavior can only be triggered by invoking some target methods/functions with some specific arguments.

$$tc \stackrel{\text{def}}{=} \langle s_1, s_2, ..., s_n \rangle \tag{2.1}$$

For simplicity, a CITRUS test case has a linear execution flow with no branching (i.e., it does not have any control statement (e.g., `if`, `for`, `while`)). Consequently, each statement s_i can be either one of the following:

1. Declaration a primitive type variable with initialization. For example, the statement "`int intVar1 = 7;`" declares a primitive `int`-typed variable `intVar1` with initial value = 7;
2. Invocation of a method or a constructor. For example, the statement "`const ClassA &objA1 = ClassA(intVar1)`" invokes a constructor `ClassA` with argument `intVar1` to construct a `ClassA`-typed object; similarly the statement "`objA1.method1(objB1)`" invokes `method1` on `objA1` with argument `objB1`.

Thus, s_i has the following characteristics:

1. Each statement has a particular *type* with zero or more *type modifiers*.
2. Each statement (except a method call whose return type is `void`) has a variable with a unique name within a CITRUS test case.
3. Every variable is assigned exactly once (i.e., static single assignment (SSA)).

2.3.2 CITRUS Test Case Example

To clearly illustrate each statement characteristic, the following example demonstrates a CITRUS test case which executes a target method `method1` on an instance of `ClassA`.

```
1:  int intVar1 = 7;
2:  ClassA objA1 = ClassA(intVar1);
3:  const ClassB objB1 = ClassB();
4:  objA1.method1(objB1);
```

The invocation of `method1` happened at line 4, right after the *object* construction of `objA1` (lines 1–2) and the *argument* construction `objB1` for `method1` (line 3). Each line corresponds to an individual statement whose characteristic is listed in Table 2.1. Note that the last statement (line 4) has *no* variable name due to its return type (i.e., `void`).

Table 2.1: Example of CITRUS Test Case and its Statement Characteristics

Line	Statement	Stmt. Type	Type	Modifiers	Var. Name
1	`int intVar1 = 7`	Prim. Decl	`int`	–	`intVar1`
2	`ClassA objA1 = ClassA(intVar1)`	Invocation	`ClassA`	–	`objA1`
3	`const ClassB objB1 = ClassB()`	Invocation	`ClassB`	`const`	`objB1`
4	`objA1.method1(objB1)`	Invocation	`void`	–	–

2.4 Process of CITRUS

This section elaborates the detailed process of test case generation in CITRUS. As depicted in Figure 2.1, CITRUS operates in the following three major stages as follows:

1. Creating the program representation of a target program.
2. Executing the method call sequence generation.
3. Post-processing CITRUS test suite.

The following subsections further explains each of the major stages in CITRUS.

2.4.1 Creating Program Representation

As the initial stage, CITRUS collects all necessary information from a target program source files (i.e., `.cpp`) through traversing abstract syntax trees (AST) of the target program. Those necessary information are described as follows:

- Lists of `class`es, `struct`s, `enum`s, and global functions declared in the target program.
- A list of header files (i.e., `.h`, `.hpp`). The collected header file information serves as an important prerequisite to compile the temporary drivers, `libfuzzer` harness drivers, and as well as the final test suite.

Then, CITRUS mines type information from the information obtained at the AST Traversal stage. CITRUS builds a type system `TS` for `class`es, `struct`s, `enum`s, and member/global functions of the target program. Algorithm 1 describes how CITRUS builds the type system `TS`. Also, CITRUS constructs an inheritance tree model (ITM) (L3–L6, L16) to take account *all* class inheritance relationship for subclass instantiation during the method sequence generation. More formally, the ITM supports CITRUS to construct only the relevant type $z \in \{C\} \cup \texttt{Subclass}(\texttt{C})$ while resolving for a class C.

Object Creators and Static Factories

CITRUS (`RegisterFunc` at L10) distinguishes "object creators" from the regular functions (other method sequence generation techniques [2, 26] apply a similar approach). Any function f that returns a non-primitive type C where $C \notin \texttt{ArgTypes}(f)$ is recognized as object creator of class C. Constructors and *static factory* methods are two most-common object creators in object-oriented programming. Also, CITRUS (`RegisterClass` at L7) registers *implicit* object creators for applicable `class`es and `struct`s, such as implicitly-declared default constructors [27] and `struct` initialization list [28].

Algorithm 1: Creating Program Representation

Data: `classes`, `enums`, `glob_fns` from AST traversal

Result: Inheritance tree model ITM and initialized type system TS

```
1  TS ← ∅; ITM ← ∅;
2  foreach cls in classes do
3      if cls has parent then
4          par ← Parent(cls);
5          ITM ← ITM ∪ {cls, par}
6      end
7      TS.RegisterClass(cls);
8      foreach m in Methods(cls) do
9          if m has public access then
10             TS.RegisterFunc(m)
11         end
12     end
13 end
14 foreach e in enums do TS.RegisterEnum(e);
15 foreach fn in glob_fns do TS.RegisterFunc(fn);
16 TS.RegisterInheritanceTreeModel(ITM);
17 repeat
18     TS.ExcludeUnsatisfiableFunctions();
19 until All fn in TS have satisfiable arguments;
```

The following paragraph concretely describes object creators and static factory methods distinguished by CITRUS's `RegisterFunc` at line 10. Listing 2.5 shows the type declarations of `class`-type `Point` (L1–L7) and a `struct`-type `Line` (L9). While most class instances are usually constructed through their public and explicitly-declared constructor functions, both `Point` and `Line` possess more non-ordinary ways to construct. In Listing 2.5, there are *four* object creators of class `Point`, which are: (1) two *static factory* methods `FromCartesian` and `FromPolar` at line 3–4; (2) an *implicit* default constructor (although its declaration was unspecified); and (3) an external function `RandomPosition` at line 10. Note that static factory methods are `public` and `static` methods that constructs and returns an instance of a particular class. Also note that although `RandomPosition` is not a member function of class `Point`, the `RandomPosition` function can still be considered an object creator of `Point` since it constructs and returns `Point` instances. Meanwhile, most `struct` types have their constructors unspecified, such as `Line` (L9) in Listing 2.5. For such `struct` types, CITRUS can obtain an instance of `Line` using the C++'s initializer list constructing expression. For example, the expression "`Line{&point1, &point2}`" is to construct a `Line` using the addresses of `point1` and `point2` as `start` and `end` respectively.

Unsatisfiable Functions

At L17–L19, CITRUS excludes all unsatisfiable functions from the list of functions. In the current implementation of CITRUS, a function f is defined as an *unsatisfiable* function if there exists a type $t \in \texttt{ArgTypes}(f)$, where t is unconstructable by CITRUS. Similarly, a member method `m` owned by class `C` is unsatisfiable if CITRUS cannot construct an instance of class `C` (i.e., because `m` can only be invoked

Algorithm 2: Method Call Sequence Generation

Data: Initialized type system TS and time budget T_{MAX}

Result: Q_{valid} and Q_{crash}: queues of valid and crashing test cases, respectively

```
1  Q_valid ← ∅; Q_crash ← ∅; Cov ← ∅; STraces ← ∅;
2  T_start ← Now();
3  while ElapsedTime(T_start) < T_MAX do
4      tc ← LoadOrGenerateTestCase(TS, Q_valid);
5      tc ← MutateTC(tc);
6      exe, err ← BuildTempExe(tc);
7      if err = ∅ then                                   /* Build successful */
8          ret_code ← Execute(exe);
9          if ret_code = 0 then                          /* Exited normally */
10             cov_tc ← MeasureCoverage(tc);
11             cov_new ← cov_tc − Cov;
12             if cov_new ≠ ∅ then
13                 Cov ← Cov ∪ cov_tc;
14                 Q_valid ← Q_valid ∪ {tc};
15             end
16         else                                          /* Crash detected */
17             out_gdb ← ExecuteInGDB(tc);
18             st_trace ← ParseStackTrace(out_gdb);
19             if st_trace not in STraces then
20                 STraces ← STraces ∪ {st_trace};
21                 Q_crash ← Q_crash ∪ {tc};
22             end
23         end
24     end
25 end
```

on an instance of `C`). Some examples of unconstructable types are as follows:

- Classes with no recognized object creators. For example, a class Y has only one constructor that takes a function pointers as argument. Class Y is *unconstructable* because function pointers are still unsupported by CITRUS.
- STL classes which are yet not handled by CITRUS, such as `thread`, `mutex`, and `function`.

2.4.2 Method Call Sequence Generation

Algorithm 2 describes the method call sequence generation (i.e., randomly generating CITRUS test cases formed by arbitrary method call sequences) to create various CITRUS test cases. The sequence is obtained by calling `LoadOrGenerateTestCase` (L4) followed by `MutateTC` (L5). If a method call sequence tc is generated, CITRUS builds tc as an executable file `exe` (L6) through compilation and linking with the target program's object files (`.o`). If the build was successful, CITRUS processes `exe` as follows:

Listing 2.5: Illustration of Object Creators and Static Factories

```
1: class Point {
2: public:
3:  static Point FromCartesian(double x, double y);
4:  static Point FromPolar(double r, double th);
5: private:
6:  double x, y;
7: };
8: Point RandomPosition();
9: struct Line { Point *start, *end; };
```

Algorithm 3: `LoadOrGenerateTestCase`

Data: Type system TS and queue of valid TCs Q_{valid}

Result: A candidate test case tc to be executed

1 $b_{gen\_new} \leftarrow$ `RandInt`$(0, 1)$; `/* 50% prob */`
2 **if** Q_{valid} *is empty* ***or*** $b_{gen\_new} == 0$ **then**
3 $funcs \leftarrow$ `AllFunctions`(TS);
4 $f_{target} \leftarrow$ random function selected from $funcs$;
5 $tc \leftarrow$ `GenTCForMethod`(f_{target});
6 **else**
7 $tc \leftarrow$ `RoundRobinSelection`(Q_{valid});
8 **end**
9 **return** tc

- If `exe`'s execution terminates normally, CITRUS will store tc into the valid queue Q_{valid} in a case that `exe` increases coverage (i.e., a new unique program behavior is induced). However, tc will be discarded if it does not increase the test coverage.
- If `exe` crashes, CITRUS re-executes `exe` in the `gdb` environment to collect the crash information such as a stack trace. Then, it puts tc into Q_{crash} if the stack trace has not been generated previously (i.e., a new crash error occurs).

The process (L3–L25) is continued until the given time budget T_{MAX} is completely consumed.

Algorithm 3 (`LoadOrGenerateTestCase`) describes how CITRUS reuses test cases from Q_{valid} during the random method call sequence generation (i.e., L4 in Algorithm 2). `LoadOrGenerateTestCase` performs either one of the following:

- Generating a new sequence from scratch (L3–L5); or
- Reusing the existing tcs from Q_{valid} in a round robin manner (L7).

The following subsections further elaborate the following core processses of generating test cases:

- How CITRUS generates a test case from scratch (`GenTCForMethod` at L5 in Algorithm 3).
- How CITRUS generates diverse test cases by mutating a test case (`MutateTC` at L5 in Algorithm 2).

Algorithm 4: `GenTCForMethod`

Data: A target function f

Result: A test case tc that calls f

```
1  stmts ← ⟨⟩; args ← ⟨⟩;
2  foreach type_arg in ArgTypes(f) do
3  |   op_arg ← ResolveType(type_arg, stmts);
4  |   args ← args · ⟨op_arg⟩;
5  end
6  if f needs invoking object then
7  |   cls_f ← ClassOwner(f);
8  |   op_inv ← ResolveType(cls_f, stmts);
9  |   s_call ← CallWithInvokingObj(f, op_inv, args);
10 else
11 |   s_call ← Call(f, args);
12 end
13 stmts ← stmts · ⟨s_call⟩;
14 return MakeTC(stmts)
```

Test Case Generation from Scratch

Algorithm 4 (`GenTCForMethod`) describes the process of test case generation from scratch (L5 in Algorithm 3). For a randomly selected target function f (L4 in Algorithm 3), CITRUS generates statements to construct f's arguments as described at L1–L5. Note that function `ResolveType` at L3 takes the desired type $type_{arg}$ as argument, and returns a variable op_{arg} (L3) that is obtained from either one of the following:

1. A variable name of an existing statement $s \in stmts$ where $\texttt{Type}(s) = type_{arg}$. Note that, to be able to reuse an existing statement s, the position of s shall precede of the position where op_{arg} will be used.

2. Construction of another sequence seq' of method calls (including all supporting statements to provide primitive arguments) that constructs a new statement s' (i.e., $s' \in seq'$) where $\texttt{Type}(s') = type_{arg}$. In this way, op_{arg} will use the variable name that correspond to the statement s', while the new sequence seq' will also be appended to the main sequence $stmts$.

When the target function f is a non-`static` member function of a particular class (L6–L9), CITRUS resolves the target object (denoted by op_{inv} at L8), on which f to be invoked. Finally, CITRUS constructs a call statement s_{call} (L9 and L11) and appends s_{call} to $stmts$.

Test Case Mutation

Algorithm 5 (`MutateTC`) describes how CITRUS mutates a CITRUS test case. CITRUS performs n-stacked modifications on a given tc (where $n \leq MAX$) to obtain a new mutated test case tc'. This is also known as *havoc* strategy in AFL [6]. To prevent the occurrences of uncompilable test case during mutation, CITRUS ensures every test case mutations to preserve the type validity in the sequence

Algorithm 5: `MutateTC`

Data: A CITRUS test case tc to mutate, MAX: a maximum number of mutations to tc
Result: The mutated test case tc'

```
1  tc' ← tc;
2  n ← RandInt(0, MAX);
3  for i ← 1 to n do
4      switch RandInt(0, 2) do
5          case 0 do
6              tc' ← Randomly insert a random method call at a random position in tc'
7          case 1 do
8              tc' ← Randomly mutate a statement in tc'
9          case 2 do
10             tc' ← Delete unused variables in tc'
11     end
12 end
13 return tc'
```

generated. More formally, for each *original* and *mutated* test case pair (tc, tc'), CITRUS ensure that both tc and tc' are syntactically valid for each test case mutating operation.

CITRUS performs three types of test case mutations: insertion, deletion, and modification as described at L5–L10. For statement modification, it performs the following six statement mutation operators as follows (CITRUS uses the same mnemonic names of mutation in Agrawal et al. [29]):

1. CGCR (Constant Replacement using Global Constant),
2. VLSR (Mutate Scalar References using Local Scalar References),
3. VLTR (Mutate Structure References using only Local Structure References),
4. CLSR (Constant for Scalar Replacement using Local Constants),
5. OAAN (Arithmetic Operator Mutation), and
6. OANG (Arithmetic Operator Negation).

2.4.3 Post-processing a CITRUS Test Suite

Finally, CITRUS stores CITRUS test cases in Q_{valid} (Algorithm 2) in the following two different formats:

1. `libfuzzer`-compatible test cases:
   CITRUS applies `libfuzzer` to the CITRUS test cases to increase test coverage further. In contrast to the method call sequence generation of CITRUS (i.e., diversifying a state of a target program P through various sequences of the method calls), `libfuzzer` alters the states of P by randomly generating various inputs to P.
2. Google Test-compatible test cases:
   The Google Test-compatible test cases is provided to integrate the CITRUS test cases with the existing Google test suite in the target program (if any).

Additionally, CITRUS outputs the de-duplicated crashing test cases stored in Q_{crash}. Crashing test cases generated by CITRUS are annotated with: (1) `gdb` stack trace output and (2) a *comment* to point at the crashing line. By these additional information, CITRUS helps the user to identify the root cause of the crash (see Section 3.3.1 for Case Study) without re-running the test case.

2.5 Crash De-duplication in CITRUS

Since crash de-duplication task is essential to reduce the effort of the time-consuming manual bug analysis, CITRUS uses stack hashes to triage crashes [30]. To generate stack hashes, CITRUS extracts a sequence of source locations (i.e., file names + line numbers) in the function call stack, which are parsed from the `gdb` stack trace output. CITRUS uses the source code locations (instead of binary code locations) because the binary code locations might be inconsistent among different runs due to constantly changing executable file during method call sequence generation. Note that CITRUS considers only the source locations in the target project directory to avoid duplicated crashes caused by uncontrolled behaviors of external library function calls.

2.6 Implementation

I have implemented CITRUS in 8.9 KLoC using modern C++. CITRUS utilizes LLVM's LibTooling framework to preprocess and parse C++ source code files. This CITRUS implementation have been tested working on Ubuntu 16.04 and later LTS versions with LLVM 11.0.1. At the time of writing, CITRUS supports the C++14 standard and I am working to support the C++17 and C++20 language features. Note that CITRUS targets programs/libraries that generate object files (`.o`) and GCOV log files (`.gcno`) during the build process. These files are necessary to build executables and measure the coverage of the target program during the testing process.

To begin testing, CITRUS requires the following items:

1. A C++ source code file u_{cpp}.
2. A compilation database (i.e., `compile_command.json`) emitted by C++ build tools (e.g., CMake, Bear) while building the target program.
3. A linking configuration to generate an executable file.

The compilation database helps CITRUS extract the compilation flags that were used for preprocessing and compiling u_{cpp}. However, such compilation databases provide no information about the necessary object files to build an executable file. To mitigate this, the current version of CITRUS requires the user to specify the linking configuration, which covers: (1) the directory where the target program's object files (`.o`) exist, and (2) additional external libraries linking flags (if any, e.g. `-lz` to use the `zlib` library). [1]

CITRUS uses LCOV to measure the testing coverage. To support this, CITRUS requires the target program's binaries to be instrumented for coverage analysis (i.e., compiled using `--coverage` flag). However, due to the C++ exception feature, coverage instrumentation on C++ programs generates too many (almost) unreachable `throw` branches (e.g., during C++ object construction) that are (almost)

[1] A future version of CITRUS will automatically capture the linking configuration by using a wrapper of a linker during the build process of a target program

Table 2.2: Type Categories in CITRUS

Type Name	Representation	Examples
`PrimitiveType`	Primitive types	`int`, `void`, `bool`
`ClassType`	C++ records (`class`/`struct`)	`class JsonValue`
`EnumType`	`enum` variants	`Color::Red`
`STLType`	`STL` classes	`std::tuple`, `std::map`
`TemplateTypenameType`	Free template type variable	`T`, `K`, `V`
`TemplateTypeSpcType`	Specialization of template class	`std::vector<int>`, `Parser<std::string>`

Table 2.3: Statement Variants in CITRUS

Statement Name	Representation	Examples
`PrimitiveStmt`	Simple assignment	`int int1 = 5;`
`ArrayInitStmt`	Array initialization	`char[2] char2 {'a','b'};`
`CallStmt`	Function/constructor call	`ClassA classa3{char2};` `int int4 = class3.Invoke(int1);`
`STLStmt`	`STL` object construction	`std::map<int,int> map5 {{1, 2}};` `std::array<int,2> {0, 0};`

never executed. To mitigate such issue, CITRUS utilizes a modified version of LCOV [31] to exclude such virtually unreachable branches.

2.6.1 Types and Statements in CITRUS

CITRUS recognizes *six* type categories in C++ language as described in Table 2.2. Note that each type can be combined with (*zero* or more) *type modifier*s to be more accurately represent the types in C++ language. For example, "`const int* const`" is a `PrimitiveType int` combined with *three* modifiers, which are: `const`, `const`-on-pointer, and pointer. Currently, there are *seven* variants of type modifiers in CITRUS: `const`, `unsigned`, pointer (`*`), array (`[]`), `const`-on-pointer, `lvalue` reference (`&`), and `rvalue` reference (`&&`).

CITRUS generates method call sequences composed from *four* kinds of statements as described in Table 2.3. Note that each statement in Table 2.3 corresponds to a type and a variable name as discussed at Section 2.3.

2.6.2 Testing C++ Template Classes/Functions

Testing template classes in C++ is challenging because it is almost impossible to instantiate template classes with all possible types. In addition, inappropriate type instantiation of template classes may generate many uncompilable test cases, which hinders the testing effectiveness. Listing 2.6 demonstrates a simple scenario in which inappropriate type instantiation produces test case that cannot be compiled. At L12, `Outer<char>` can be compiled because `Inner(char*)` is defined. Instantiating `Outer<int>` at

Listing 2.6: Challenge in Instantiating Template Classes in C++

```
 1: class Inner {
 2:   public: Inner(char *a) {}
 3: };
 4: template<typename T>
 5: class Outer {
 6:   public: Outer(T* t) {
 7:       const Inner &tmp = Inner(t);
 8:   }
 9: };
10: void Decode(Outer<char> &arg) { ... }
11:
12: Outer<char> ok((char*) nullptr); /* OK */
13: Outer<int> err((int*) nullptr); /* FAIL */
```

L13, however, causes a compilation error because `Inner(int*)` is not defined.

To reduce the number of uncompilable test cases generated, CITRUS conservatively binds a free type variable `T` to a concrete type according to the existing type bindings in the target program. For example, CITRUS binds `T` to `char` (denoted as $\{\texttt{T} \mapsto \texttt{char}\}$) when it generates a method call sequence for `Decode` at L10 because the argument requires `Outer<char>&` type. However, such type hinting may not always be available, such as when CITRUS targets the constructor of `Outer<T>` at L6. In this case, CITRUS randomly selects from either $\{\texttt{T} \mapsto \texttt{int}\}$ or $\{\texttt{T} \mapsto \texttt{double}\}$. All of these type bindings are stored within a context map W : `name` $\rightarrow$ `type`. In CITRUS, the context map W is implemented as `TemplateTypeContext` class, whose declaration can be found at `include/type.hpp` header file.

2.6.3 Handling C++ STL Classes

C++ supports various useful STL classes and most C++ programs frequently utilize the STL classes. However, automated unit-level testing on C++ programs that heavily use STL classes has several technical challenges, such as:

1. Some STL classes have more *indirect* ways of construction, rather than by simply constructor-calling convention. For example, `unique_ptr` and `tuple` should be constructed through `make_unique` and `make_tuple` API respectively, instead of their constructors.

2. Most STL classes contain many member functions, and including all of such functions in method sequence generation may not contribute to exploring diverse program states. For example, CITRUS may generate `vector` objects (with arbitrary sizes and elements) by using the `push_back` API only. Thus, generating random sequences of method calls with `resize` and `erase` operators on `vectors` is ineffective towards exploring new executions (i.e., just enlarging the search space).

CITRUS mitigates such technical challenges by putting additional engineering efforts to handle the construction of each STL class without performing random sequence generation. When a function f requires an object of a STL class $t_{\texttt{STL}}$ as an argument, CITRUS constructs an *initialized* $t_{\texttt{STL}}$-typed object $o_{\texttt{STL}}$ by using a *single-line* STL construction statement. For example, CITRUS uses C++ *initializer list* syntax [32] to construct objects of STL containers. This way, CITRUS does not need to generate method call sequence to construct arbitrary STL objects of the STL classes.

Currently, CITRUS handles 23 STL classes in the four categories as follows [2]:

- Containers (e.g., `vector`, `set`, `map`, `forward_list`).
- Utility (e.g., `pair` and `tuple`).
- Strings (e.g., `basic_string`, `string`, `wstring`).
- Memory (e.g., `unique_ptr` and `shared_ptr`).

[2] The complete list of the STL classes that CITRUS supports can be found at `include/type.hpp` header file.

Chapter 3. Evaluation

3.1 Experiment Setup

3.1.1 Research Questions

I have developed the following two research questions to evaluate CITRUS:

RQ1: How effective is CITRUS in terms of branch coverage? To what extent does CITRUS achieve test coverage on the eight target subjects in Table 3.1? To answer RQ1, I allocated 12 hours to perform method call sequence generation and two minutes `libfuzzer` fuzzing time for each test case generated.

RQ2: How effective and efficient is CITRUS compared to the other CITRUS variants? In which configuration does CITRUS achieve the highest test coverage? To answer RQ2, I introduced four CITRUS variants by assigning different time budgets for the method call sequence generation stage and `libfuzzer` fuzzing stage.

3.1.2 Target Subjects

I applied CITRUS on the eight popular real-world C++ programs in Table 3.1 ranging from 1.5KLoC (`jsonbox`) to 20KLoC (`re2`). I specifically targeted only C++ libraries (not C) to demonstrate that CITRUS works on the complex C++ language features, such as polymorphism, template classes, STL types, and so on. Most subjects are parsing libraries because parsing is one important stage on program execution, and therefore it requires to be extensively tested. Note that since CITRUS requires object files (`.o`) and GCOV log files (`.gcno`) to perform testing, all libraries in Table 3.1 are not *header-only* libraries.

Table 3.2 summarizes the statistics of accessible functions in each of the target subjects. Note that only the number of accessible functions was considered since CITRUS does not directly invoke `private` methods. As mentioned in Section 2.4, some unsatisfiable functions were excluded due to unconstructible argument types (see the 3rd and 4th column). I noticed that there were larger number of unsatisfiable functions in `tinyxml2` and `json-voorhees`, and I have investigated the reasons of such unsatisfiable functions in these two subjects:

- For `tinyxml2`, 158 functions were unsatisfiable because CITRUS failed to recognize the non-static member functions (that returns a particular class) for constructing one of its argument types [1].
- For `json-voorhees`, 111 functions were due to argument dependency to one of unhandled STL classes (e.g., `std::type_index`, `std::type_info`); 83 functions were due to *type aliasing* [33] within template classes, which are still unhandled by CITRUS; and 28 functions were due to dependency with *other* unsatisfiable functions (which had been excluded).

[1] CITRUS do not utilize such non-static functions as object creators as they do not always perform object construction. For instance, it is common to write a setter method that returns `this` reference for method chaining, such as: "`Point *SetX(int x) { x_ = x; return this; }`" (which performs no object construction although it returns a `Point` reference).

Table 3.1: Target Subjects in CITRUS Experiment

Name	Size (LoC)	Commit Hash	URL
jsonbox	1,477	6f86f81	github.com/anhero/JsonBox.git
hjson	2,911	0c40199	github.com/hjson/hjson-cpp.git
tinyxml2	3,606	1dee28e	github.com/leethomason/tinyxml2.git
jvar	4,860	e2a6a43	github.com/YasserAsmi/jvar
jsoncpp	5,420	c39fbda	github.com/open-source-parsers/jsoncpp.git
json-voorhees	8,614	046083c	github.com/tgockel/json-voorhees.git
yaml-cpp	8,800	b591d8a	github.com/jbeder/yaml-cpp.git
re2	20,373	bc42365	github.com/google/re2.git

Table 3.2: Accessible Function Statistics in Target Subjects

Subject	#Total Public Func. (TF)	#Unsatisfiable Func. (UF)	%UF (#UF/#TF)
jsonbox	89	16	17.98
hjson	181	12	6.63
tinyxml2	345	166	48.12
jvar	344	28	8.14
jsoncpp	211	11	5.21
json-voorhees	654	233	35.63
yaml-cpp	535	95	17.76
re2	427	63	14.75

- The remaining unsatisfiable functions (8 in `tinyxml2` and 11 in `json-voorhees`) were mostly caused by unhandled argument types by CITRUS, such as: multi-dimensional pointers, opaque pointers (`void*`), and `FILE`.

3.1.3 CITRUS Variants

To answer RQ2, I used four CITRUS variants to compare. Those four CITRUS variants are as follows:

- $\texttt{C}_{12+\texttt{LF2}}$, which generates sequences of method calls for 12 hours and applies `libfuzzer` for two minutes for each test case generated. This is the main configuration of CITRUS.
- $\texttt{C}_{6+\texttt{LF1}}$, which generates sequences of method calls for six hours and applies `libfuzzer` for one minute for each test case generated.
- $\texttt{C}_{6+\texttt{LF3}}$, which generates sequences of method calls for six hours and applies `libfuzzer` for three minutes for each test case generated.
- $\texttt{C}_{24}$, which generates sequences of method calls for 24 hours.

3.1.4 Environment Setup

I conducted CITRUS experiments in SWTV-Lab's cluster in which each node is equipped with Intel Core i5-4670K CPU (3.4 GHz), has 16GB RAM, and running Ubuntu 16.04 64-bit version. To reduce the random variance caused by the randomized algorithm, I only reported the averaged result of the experiments collected from ten repeated runs. For statistical measure, I also have reported the standard deviation of the experiments, and showed that CITRUS produced *consistent* results (i.e., low variance) from the 10 repeated runs experiment (see Table 3.5). I used $MAX = 20$ as the maximum number of mutation operations.

Most libraries at Table 3.1 consist of multiple source code files, while the current version of CITRUS can only take *one* source code file (`.cpp`) at a time. For such cases, CITRUS can be run on each individual source code file (and generate test cases for each file). However, running CITRUS this way requires *dedicated* time budget for each individual target source code file, which may not be an effective time-budget allocation since not all source code files shares a similar complexity (e.g., some files may contain more functions/branches than others). Therefore, to run CITRUS targeting multi-files C++ programs using a single time budget in this experiment, for each subject I used a *proxy* source code file named "`all.cpp`" that inherently imports all header files of the target program. By targeting solely the `all.cpp` file, CITRUS targets the whole library in a single run (i.e., without requiring to specify individual time budget for each `.cpp` file).

3.1.5 Threats to Validity

The possible threat to external validity is the generality of subject selection. To reduce the external validity risk, the target subjects selected for CITRUS experiment consist of C++ open-source programs of varying sizes. The threat to internal validity is bug existence in CITRUS implementation. To reduce the internal validity risk, I carefully wrote and extensively tested CITRUS implementation. The construct threat to validity is the appropriate implementation in CITRUS' dependencies (e.g., LLVM/Clang, LCOV). To reduce the construct validity risk, I implemented CITRUS with sufficiently recent versions of dependencies.

3.2 Experiment Results

3.2.1 Statistics on CITRUS Test Cases

Table 3.3 shows the statistics of test cases generated by $\texttt{C}_{\texttt{12+LF2}}$ on the eight target subjects. For example, for `jsonbox`, CITRUS ($\texttt{C}_{\texttt{12+LF2}}$) generated 116.2 test cases each of which is 33.6 lines long on average over the ten repeated experiment runs (see the second row of the table).

3.2.2 RQ1: How effective is CITRUS ($\texttt{C}_{\texttt{12+LF2}}$) applied on the target programs?

The experiment result shows that the test cases generated by CITRUS achieve high statement coverage and branch coverage. Figure 3.1 shows the statement coverage and the branch coverage obtained by the test cases generated by CITRUS ($\texttt{C}_{\texttt{12+LF2}}$) on the eight target subjects. CITRUS achieved roughly 80% or higher statement coverage in 87.5% (=7/8) of all target subjects (i.e., in all target subjects except

Table 3.3: Statistics of the Test Cases Generated by $\texttt{C}_{12+\texttt{LF2}}$ on the Target Subjects

Subject	# Test Cases			Length of a Test Case (LoC)			
	avg	**min**	**max**	**avg**	**stdev**	**min**	**max**
jsonbox	116.2	101	125	33.6	21.2	3	132
hjson	237.6	219	254	26.3	15.7	3	118
tinyxml2	136.5	125	145	41.7	25.0	4	156
jvar	209.9	200	219	30.9	20.0	2	137
jsoncpp	283.2	274	291	29.8	19.6	2	133
json-voorhees	240.5	224	263	21.8	14.1	2	116
yaml-cpp	141.1	132	155	25.0	16.1	3	85
re2	303.6	292	322	39.5	27.8	3	174

Table 3.4: Time Cost for CITRUS Variants

Subject	$t_{\texttt{total}}$ (h)			
	$\texttt{C}_{24}$	$\texttt{C}_{6+\texttt{LF1}}$	$\texttt{C}_{6+\texttt{LF3}}$	$\texttt{C}_{12+\texttt{LF2}}$
jsonbox	24	8.0	12.1	16.2
hjson	24	9.8	17.5	20.5
tinyxml2	24	8.2	12.5	16.8
jvar	24	9.6	16.7	19.3
jsoncpp	24	10.5	19.5	21.7
json-voorhees	24	10.0	18.1	20.8
yaml-cpp	24	8.2	12.7	17.2
re2	24	10.6	19.9	22.7
Average	24	9.4	16.1	19.4

tinyxml2). For jsonbox and jsoncpp subjects, it even achieved >90% statement coverage. For branch coverage, CITRUS achieved roughly 60% or higher in the majority of the target subjects (=6/8) (i.e., in all target subjects except tinyxml2 and json-voorhees).

Answer to RQ1: On the eight real-world C++ target programs, CITRUS shows high testing effectiveness in terms of both statement and branch coverage (i.e., it achieved 50% to 95% statement coverage and 40% to 79% branch coverage).

3.2.3 RQ2: How effective and efficient is CITRUS compared with other CITRUS variants?

Table 3.5 shows the coverage achieved by the four CITRUS variants (i.e., $\texttt{C}_{24}$, $\texttt{C}_{6+\texttt{LF1}}$, $\texttt{C}_{6+\texttt{LF3}}$, and $\texttt{C}_{12+\texttt{LF2}}$). The highest coverage values achieved are shown in bold font. Note that the running time of each variant (except $\texttt{C}_{24}$) may vary depending on how many test cases were constructed during the method call sequence generation part of CITRUS. The running time of each variant is summarized at

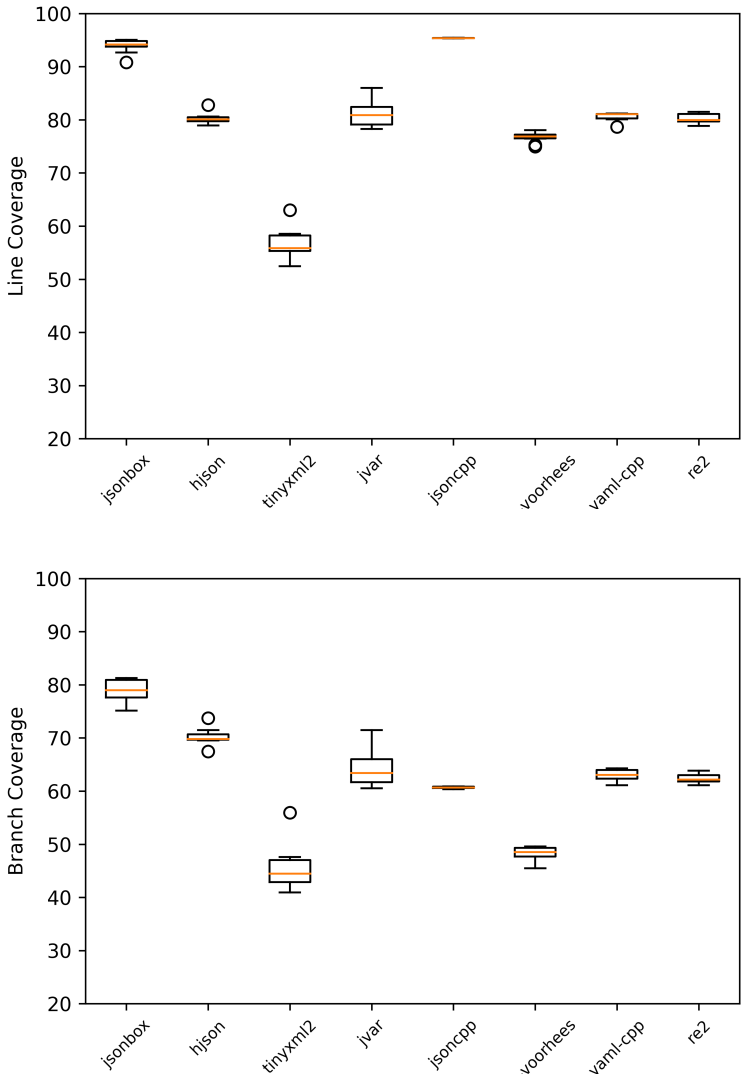

Figure 3.1: Statement and Branch Coverage Achieved by CITRUS ($\texttt{C}_{12+\texttt{LF2}}$) on the Target Programs.

Table 3.4.

Table 3.5 shows that, among the four CITRUS variants, $\texttt{C}_{12+\texttt{LF2}}$ (the main configuration of CITRUS) achieved the highest statement coverage (80.6%), the highest branch coverage (61.7%), and the highest function coverage (75.4%) on average over the eight target programs (see the last row of the table). For example, $\texttt{C}_{12+\texttt{LF2}}$ achieved the highest statement coverage on the five out of the eight target programs (i.e., all the target programs except `jsonbox`, `tinyxml2`, and `jvar`) (see the fifth column of the table), the highest branch coverage on the six out of the eight target program (i.e., all except `tinyxml2` and `jvar`), and the highest function coverage on the five out of the eight target programs.

Table 3.4 shows the time cost consumed by the four CITRUS variants. All three CITRUS variants that utilize `libfuzzer` spent less time than that of $\texttt{C}_{24}$. In particular, $\texttt{C}_{12+\texttt{LF2}}$ spent 19%(=(24-19.4)/24) less time than $\texttt{C}_{24}$, but achieved 12% higher (=(80.6-72.0)/72.0) statement coverage, 13% higher (=(61.7-54.6)/54.6) branch coverage, and 5% higher (=(75.4-72.1)/72.1) function coverage on average over all target programs (see the last row of Table 3.5).

As a result, from Table 3.4 and Table 3.5, I can confirm that the idea of integrating `libfuzzer` into

CITRUS is effective and efficient to increase test coverage. This is because since the CITRUS variants that utilize `libfuzzer` (i.e., $\texttt{C}_{6+\texttt{LF1}}$, $\texttt{C}_{6+\texttt{LF3}}$, and $\texttt{C}_{12+\texttt{LF2}}$) achieved higher statement and branch coverage than $\texttt{C}_{24}$ and the time costs of the CITRUS variants that utilize `libfuzzer` were lower than that of $\texttt{C}_{24}$.

> **Answer to RQ2:** In most subjects, CITRUS variant $\texttt{C}_{12+\texttt{LF2}}$ produced the best result. Also, applying `libfuzzer` helps CITRUS improve the coverage score and time cost, compared to the technique that uses only method call sequence generation ($\texttt{C}_{24}$ variant).

3.3 Bug Detection Capability & Crash Analysis

Bug detection is the ultimate goal of testing tools. For unit-level test case generation however, bug detection requires additional *oracles* to distinguish true positive from the false alarms (e.g., Randoop [1] uses *contract checkers* to assert only the allowed program executions). Thus, detecting real C++ bugs in unit-level has its own challenges, because:

1. The availability of oracles in real-world programs is strictly limited. Such oracles do not exist in most C++ programs/libraries; meanwhile *artificially*-creating such oracles require expertise of the respective target subject.

2. Even the heuristics to approximate the oracles (i.e., disallowed program behavior) is hard to determine for unit-level testing context. For system-level testing, crashes are widely-used heuristic to uncover new bug; however, most crashes in unit-level testing does not always correspond to a crash bug. For example, I found a bug report [2] in `tinyxml2` with `AddressSanitizer`'s violation was considered as a false alarm by the upstream developers.

For these reasons, I demonstrated the bug detection capability of CITRUS in the following subsections.

1. Section 3.3.1 explains how CITRUS is still capable to detect real crash bugs in a target subject through a crash replication case study. In this section, I applied CITRUS to check whether it could successfully discover real crash bugs in the past versions of C++ programs.

2. Section 3.3.2 elaborates kind of crashes detected by CITRUS, followed by providing some code examples of such crashes. Based on these crash types, I pointed out several suggestions to improve CITRUS on reducing the number of false positive crash reports (see Section 4.2).

3.3.1 Case Study of Crash Detection of Past Bugs

I have conducted a case study to show how CITRUS detects crash bugs in a real-world C++ program. A bug b is counted as *detectable* by CITRUS if a proof-of-concept (PoC) of crash from the exact same root cause b was identified by CITRUS. For this case study, I used the following search criteria to mine relevant patches from the `git` commit messages from all *eight* target libraries:

1. The target crash bug was reported in the `git` commit message that contains one of the following keywords: "crash", "segmentation fault", "SIG", and "SEGV".

[2] https://github.com/leethomason/tinyxml2/issues/832#issuecomment-693009339

Table 3.5: Coverage Achieved by CITRUS Variants

Subject	Avg. % Statement Coverage (± stdev.)			
	C_{24}	C_{6+LF1}	C_{6+LF3}	C_{12+LF2}
jsonbox	93.6 (±0.5)	93.7 (±1.5)	**94.2** (±1.2)	93.9 (±1.3)
hjson	70.2 (±0.6)	78.8 (±1.7)	79.7 (±1.3)	**80.2** (±1.1)
tinyxml2	**59.5** (±1.8)	52.9 (±3.8)	53.2 (±3.8)	56.6 (±3.1)
jvar	**84.5** (±3.0)	80.6 (±2.1)	80.8 (±2.1)	81.2 (±2.5)
jsoncpp	60.3 (±1.5)	59.7 (±4.4)	60.1 (±4.5)	**95.4** (±0.1)
json-voorhees	69.3 (±1.0)	74.7 (±1.4)	75.6 (±1.5)	**76.7** (±1.0)
yaml-cpp	67.2 (±2.5)	78.4 (±1.1)	79.0 (±0.8)	**80.6** (±0.8)
re2	71.1 (±1.5)	77.4 (±1.8)	79.1 (±1.4)	**80.2** (±1.0)
Average	72.0	74.5	75.2	**80.6**

Subject	Avg. % Branch Coverage (± stdev.)			
	C_{24}	C_{6+LF1}	C_{6+LF3}	C_{12+LF2}
jsonbox	75.8 (±1.0)	78.6 (±2.5)	**79.1** (±1.9)	78.9 (±2.2)
hjson	57.6 (±0.9)	68.5 (±2.1)	69.8 (±1.8)	**70.2** (±1.6)
tinyxml2	**49.1** (±2.9)	41.5 (±4.8)	42.1 (±4.6)	45.5 (±4.3)
jvar	**69.7** (±4.1)	63.7 (±3.1)	64.0 (±3.2)	64.5 (±3.8)
jsoncpp	45.3 (±1.1)	46.9 (±5.0)	47.2 (±5.0)	**60.7** (±0.2)
json-voorhees	41.8 (±1.5)	48.9 (±1.8)	**50.3** (±1.7)	48.3 (±1.3)
yaml-cpp	45.9 (±1.9)	60.9 (±1.2)	61.6 (±1.0)	**63.0** (±1.0)
re2	51.3 (±1.1)	59.3 (±1.5)	61.2 (±1.2)	**62.4** (±0.9)
Average	54.6	58.5	59.4	**61.7**

Subject	Avg. % Function Coverage (± stdev.)			
	C_{24}	C_{6+LF1}	C_{6+LF3}	C_{12+LF2}
jsonbox	**93.3** (±0.0)	92.6 (±1.5)	92.6 (±1.5)	92.6 (±1.5)
hjson	36.8 (±0.9)	37.5 (±0.4)	37.6 (±0.4)	**38.1** (±0.5)
tinyxml2	**63.8** (±1.4)	59.0 (±2.1)	59.3 (±2.6)	61.2 (±1.5)
jvar	**89.5** (±2.0)	86.9 (±1.9)	86.9 (±1.8)	87.0 (±2.4)
jsoncpp	74.2 (±0.8)	72.7 (±2.5)	72.8 (±2.5)	**95.0** (±0.1)
json-voorhees	59.9 (±0.8)	61.3 (±0.9)	61.7 (±0.9)	**64.3** (±0.5)
yaml-cpp	77.2 (±2.0)	79.3 (±1.1)	79.5 (±1.1)	**80.8** (±0.7)
re2	82.0 (±1.1)	82.4 (±1.8)	83.3 (±1.2)	**84.2** (±0.9)
Average	72.1	71.5	71.7	**75.4**

Table 3.6: Five Target Crash Bugs for the Crash Detection Study

Subject	Commit Hash	Patch Date	Commit Message	Detected
`hjson`	`e8f8693`	`2018-10-07`	Fix segfault in deep_equal comparison of empty vectors (#8)	✓
`tinyxml2`	`e8f4a8b`	`2017-09-15`	Fix crash when element is being inserted "after itself"	✓
`jsoncpp`	`f6d785f`	`2016-09-25`	Fix poss SEGV	–
`jsoncpp`	`f251f15`	`2017-01-17`	Fix crash issue due to NULL value.	✓
`yaml-cpp`	`396a970`	`2014-03-22`	Fix SEGV in ostream_wrapper	✓

```
513  513      case VECTOR:
514  514        {
515  515          auto itA = ((ValueVec*)(this->prv->p))->begin();
516  516          auto endA = ((ValueVec*)(this->prv->p))->end();
517  517          auto itB = ((ValueVec*)(other.prv->p))->begin();
518       -      do {
     518  +      while (itA != endA) {
519  519            if (!itA->deep_equal(*itB)) {
520  520              return false;
521  521            }
522  522            ++itA;
523  523            ++itB;
524       -      } while (itA != endA);
     524  +      }
525  525        }
526  526        return true;
527  527
528  528      case MAP:
529  529        {
530  530          auto itA = this->begin(), endA = this->end(), itB = other.begin();
531       -      do {
     531  +      while (itA != endA) {
532  532            if (!itA->second.deep_equal(itB->second)) {
533  533              return false;
534  534            }
535  535            ++itA;
536  536            ++itB;
537       -      } while (itA != endA);
     537  +      }
```

Figure 3.2: Crash Fix in `hjson e8f8693` commit

2. The fixed code is still available in the latest version of a target program (i.e., the patch should not be removed/modified since the original patch date).

3. The crash bug resides in the target program implementation source code (i.e., the crash fix should

```
---------------------------- src/hjson_value.cpp -----------------------
index 01a3596..ac44a78 100644
@@ -516,6 +516,7 @@ bool Value::deep_equal(const Value &other) const {
       auto endA = ((ValueVec*)(this->prv->p))->end();
       auto itB = ((ValueVec*)(other.prv->p))->begin();
       do {
+        CHECK_BUG (itA != endA && "Found Bug e8f8693");
         if (!itA->deep_equal(*itB)) {
           return false;
         }
@@ -529,6 +530,7 @@ bool Value::deep_equal(const Value &other) const {
     {
       auto itA = this->begin(), endA = this->end(), itB = other.begin();
       do {
+        CHECK_BUG (itA != endA && "Found Bug e8f8693");
         if (!itA->second.deep_equal(itB->second)) {
           return false;
         }
```

Figure 3.3: Crash Replication Environment to Detect `hjson e8f8693` Bug

be applied to a source file of the target program, not in the existing test suites, fuzzers, and so on).

4. The crash occurs due to internal logic problems, not external environment-related problems (i.e., any crash related to OS, external files, or environment variables were excluded).

5. The crash bug can be understood without deep domain expertise. Otherwise, it is unverifiable if a unit-level crashing execution of a CITRUS test case really conforms to that of the reported crash bug. [3]

Table 3.6 shows the five target crash bugs selected by the above criteria. For each target crash bug, I manually set up the unit-level crash replication environment (see Figure 3.3 for an example) to check if a crashing execution of a CITRUS test case really conforms to that of the reported crash bug. Then, I ran CITRUS three times with 12 hours of method sequence generation per each run. As a result, CITRUS detected 80% (4/5) of the target crash bugs.

The detail of how CITRUS detects the crash bug in `hjson` (see the second line in Table 3.6) is described as follows. The root cause of the fix `e8f8693` in `hjson` is an *off-by-one* error in `deep_equal` comparison on empty vectors (or empty maps). As shown in Figure 3.2, prior to `e8f8693`, `deep_equal` used `do...while` to compare the elements, causing the loop body to be still executed on empty vectors (or maps); the patch avoids the loop body execution for empty vectors (or maps) by changing `do {...} while(c)` to `while(c) {...}`.

To set up the crash environment, I reverted the `e8f8693` commit and put check conditions (i.e., `itA != endA`) to reveal *off-by-one* error, as shown in Figure 3.3. Listing 3.1 shows a crashing CITRUS test case that detects the crash bug in `hjson`. At L17, CITRUS points to the crashing line that invoke `deep_equal`. Since `value5` of `value5.deep_equal(value3)` at L18 was an empty map (see L13 and L16 where `char2` was an empty string `""`), the test case triggered the crashing bug and crashed. Also, CITRUS annotates the crashing test case with additional `gdb` backtrace information (L2–L10)

[3] A crash bug report usually provides a system-level test input to replay the target crash in system-level. To check if a unit-level test execution matches that of the crashing system-level execution, we must check if the unit-level test execution satisfies *necessary conditions* of the target crash, which requires detailed understanding of the crash and the target program.

Listing 3.1: CITRUS Test Case that Detects The Crash Reported in `hjson e8f8693` commit

```
 1: TEST(CITRUS_TestSuite, tc_id_129) {
 2: // gdb output: ...
 3: // hjson/src/hjson_value.cpp:536:
 4: //   bool Hjson::Value::deep_equal(const Hjson::Value &) const:
 5: //   ...
 6: //
 7: // Program received signal SIGABRT, Aborted.
 8: // #0 __GI_raise (sig=sig@entry=6) at .../raise.c:50
 9: // #1 0x00007ffff7a59859 in __GI_abort () at abort.c:79
10: // ... */
11:   Hjson::Value value0{0.251040};
12:   bool bool1 = value0.operator!=(0.666285);
13:   char char2[1] = "";
14:   Hjson::Value value3 = Hjson::Unmarshal(char2);
15:   char char4[5] = "h6yA";
16:   Hjson::Value value5 = Hjson::Unmarshal(char2);
17:   /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
18:   bool bool6 = value5.deep_equal(value3);
19:   Hjson::Value value7 = Hjson::Unmarshal(nullptr, 50);
20:   ... }
```

For `jsoncpp f6d785f`, CITRUS failed to discover the crash bug because the bug was located inside a method with `private` modifier and a CITRUS test case does not directly invoke `private` methods.

3.3.2 Analysis of Crashes

The following section describes how I analyzed crashing test cases on all eight subjects.

Analysis Procedure

Table 3.7 shows the statistics of crashing test cases found in all 10 repeated runs of CITRUS experiment. The *second* column of Table 3.7 represents the sum count of *all* crashing test cases from 10 runs during method sequence generation. Note that CITRUS performs automated crash de-duplication on each crashing test case it encounter within single run, as described in Section 2.5. In total, CITRUS detected total of 7,439 crashing test cases in the experiment.

To obtain the number of unique crashes de-deduplicated across the 10 runs experiments, I performed another *stack trace*-based de-duplication process from all crashing test cases CITRUS discovered on each respective subject. This *meta-deduplication* task is viable without re-executing the crashing test cases because CITRUS provides the `gdb` output as supplementary information in the finalized test suite, as depicted in Listing 3.1. The *final* column represents the number of de-duplicated crashing test cases obtained by performing *meta-deduplication* across the 10 runs experiment. In total, I found 1,103 unique crashes across all eight subjects I used in the CITRUS experiment.

For each unique crash, I manually analyzed the possible correspondence of each crash to some “bug”

Table 3.7: Statistics of Crashing Test Cases

Subject	Total Crash (10 Runs)	De-duplicated Crash
yaml-cpp	75	9
hjson	123	20
jsonbox	167	29
json-voorhees	370	54
jsoncpp	864	124
jvar	1,996	220
tinyxml2	2,004	292
re2	1,840	355
Total	**7,439**	**1,103**

in the respective target program. Each unique crash report is later classified into one of the following cases:

1. *False positive crash* if the crash does not correspond to any bug in the target program. The false positive crashes are mostly happened due to the semantically-invalid sequence generated by CITRUS.

2. *Candidate crash bug* if the crash has a chance to be induced by a *faulty* implementation in target program source code. These crashes will be reported to the upstream developers to verify the root cause of the crash.

By manually analyzing the 1,103 unique crashes produced by CITRUS, I found almost the entire crashes were false positive crashes. Four crashes were categorized as candidate crash bugs (and reported), of which only *one* crash was successfully approved by the upstream developer. Those four cases were:

1. At `yaml-cpp`, method `DecodeBase64` performed incorrect *type casting* while iterating over the `string` argument. The `DecodeBase64` induced a *segmentation fault* when invoked on a `string` with negative values [4]. This is the only one crash that has been successfully approved by the upstream developer.

2. At `jsoncpp`, infinite recursion of `dupPayload` calls occurs on copying a `Json::Value` with a *cyclic* reference (e.g., appending a `Json::Value` reference to itself using the `append(Json::Value&& value)` API [5]). However, the upstream developers argued that: (1) preventing such cases require a cycle detection mechanism that would be computationally expensive if implemented in the library layer; and (2) the cyclic reference shall be prevented by the clients in their application layer. Thus, the upstream developers decided to take no action on this particular case.

3. At `re2`, CITRUS found that mutiple calls to `Regexp::Decref` function would induce a program crash [6]. This is later clarified by the upstream developer that the `Regexp::Decref` API performs *decrement* on the `Regexp` object's reference counting, and immediately followed by *destroying* the `Regexp` object when the reference counter reaches zero. Performing multiple calls to `Regexp::Decref` triggers undefined behavior as it performs invalid address dereference starting from the second

[4] https://github.com/jbeder/yaml-cpp/pull/1051
[5] https://github.com/open-source-parsers/jsoncpp/pull/1342
[6] https://github.com/google/re2/issues/341

call (i.e., this candidate crash was not a real bug as the undefined behavior was expected from dereferencing an invalid pointer).

4. At `jvar`, CITRUS discovered that crash could occur on dereferencing an uninitialized `V_POINTER`-typed `jvar::Variant` instance [7]. The `V_POINTER`-typed `jvar::Variant` can be constructed using `Variant(Type)` constructor, but can be initialized only through the `internalSetPtr` call (i.e., unintended crashes could occur if the client dereferences the `V_POINTER`-typed `jvar::Variant` without initializing the pointer value unintentionally). Thus, for this case I suggest API changes to prevent the construction of such uninitialized type to avoid the unintended crash.

The next subsection describes each of the false positive categories in more detail.

Categories of False Positive Crashes

Figure 3.4 shows the distribution each false positive category occurrences across all subjects, while Table 3.8 lists the types of false positives found in CITRUS experiment. ∼77% (=(486+363)/1103) of false positive crashes were caused by dereferencing invalid addresses (INVA) and `NULL` values (NDRF). Some remaining false positive categories (such as DIV0, ICAS, STVR, and ARRI) happened in less than 50 occurrences in the experiment, which are less frequent compared to INVA (486 occurrences) and NDRF (363 occurrences). Based on this data, we could conclude that eliminating false positive crashes from both the INVA and NDRF categories is highly advantageous to reduce the required manual efforts while investigating crashing test cases.

Figure 3.5 describes the category distribution of false positive crashes per subject. Figure 3.5a shows the count of each false positive category appearing in each subject, while Figure 3.5b shows the proportion of every false positive category in each respective subject. From these two sub-figures, we can observe that false positive occurrences have no insightful trend (i.e., arbitrary) in each subject (which implies that each subject possesses its own characteristics). For example, $> 90\%$ false positive crashes in `hjson` came from DIV0 category, which *almost* did not occur anywhere in other subjects. The INVA and NDRF, which are two most frequent categories in the experiment, interestingly did not occur in `jsonbox` subject. Meanwhile, $\sim 80\%$ crashes in the `jsonbox` subject were induced by incorrect type casting in template specialization-typed variable (ICAS).

In the following paragraphs, I will elaborate each category of false positive crashes (also provided with a concrete example) I found from analyzing total 1,103 de-duplicated crashes in the CITRUS experiment.

Invalid address dereference (INVA). The INVA category happens when the method call sequence generated by CITRUS produced an invalid pointer (i.e., address) which is later dereferenced during the test case execution. The INVA category can always happen during method sequence generation because determining the validity of pointers in C/C++ still remains as an undecidable problem.

The INVA category was considered a false positive because invalid addresses were produced by incorrect implementation of the test case (produced by CITRUS) in most cases. This implies that such crashes did not correlate with any fault in the target program implementation, but was induced by the faulty call sequences generated by CITRUS itself. Two examples of INVA crashes in the CITRUS experiment are as follows:

[7] https://github.com/YasserAsmi/jvar/issues/34

Table 3.8: Seven False Positive Categories in CITRUS Experiment

No	Description	Mnemonic
1	Invalid address dereference	INVA
2	NULL pointer dereference	NDRF
3	Method assertion (precondition) failure	MASS
4	Invalid array index / size	ARRI
5	Passing address of stack variable to method that takes ownership	STVR
6	Invalid type casting on template specialization types	ICAS
7	Division by zero	DIV0

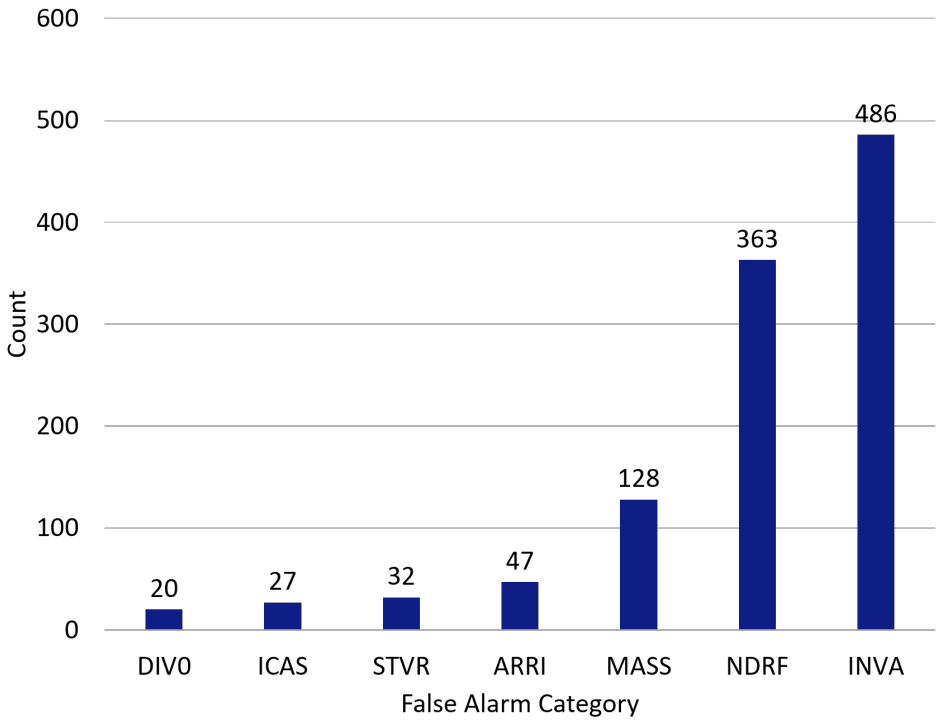

Figure 3.4: Distribution of False Positives in CITRUS Experiment

1. At `re2` (Listing 3.2), method `Prog::inst` returns an address of `inst_` (a `vector`-typed field in `Prog`) incremented with parameter `id` (B1). When CITRUS invokes `inst` using negative values as argument (A2), `inst` returns an invalid *unknown* address to be stored at `inst13`. Storing invalid addresses did not halt the program execution; but however, the test case crashed at line A5 as it attempted to invoke `empty` method on the address pointed by `inst13` (which pointed to an invalid address).

2. Iterator class pattern, which is mostly used to iterate over elements in a data structure without concerning its underlying implementation, can also trigger the INVA false positive category. At `jsoncpp` (Listing 3.3), the iterator variable `valueconstiterator1` (A2) pointed to an *unknown* address because it was a *copy* of `valueiterator0`, which was created through the default constructor API (A1) *without* specifying any collection it should point to (i.e., `valueiterator0` was pointing to

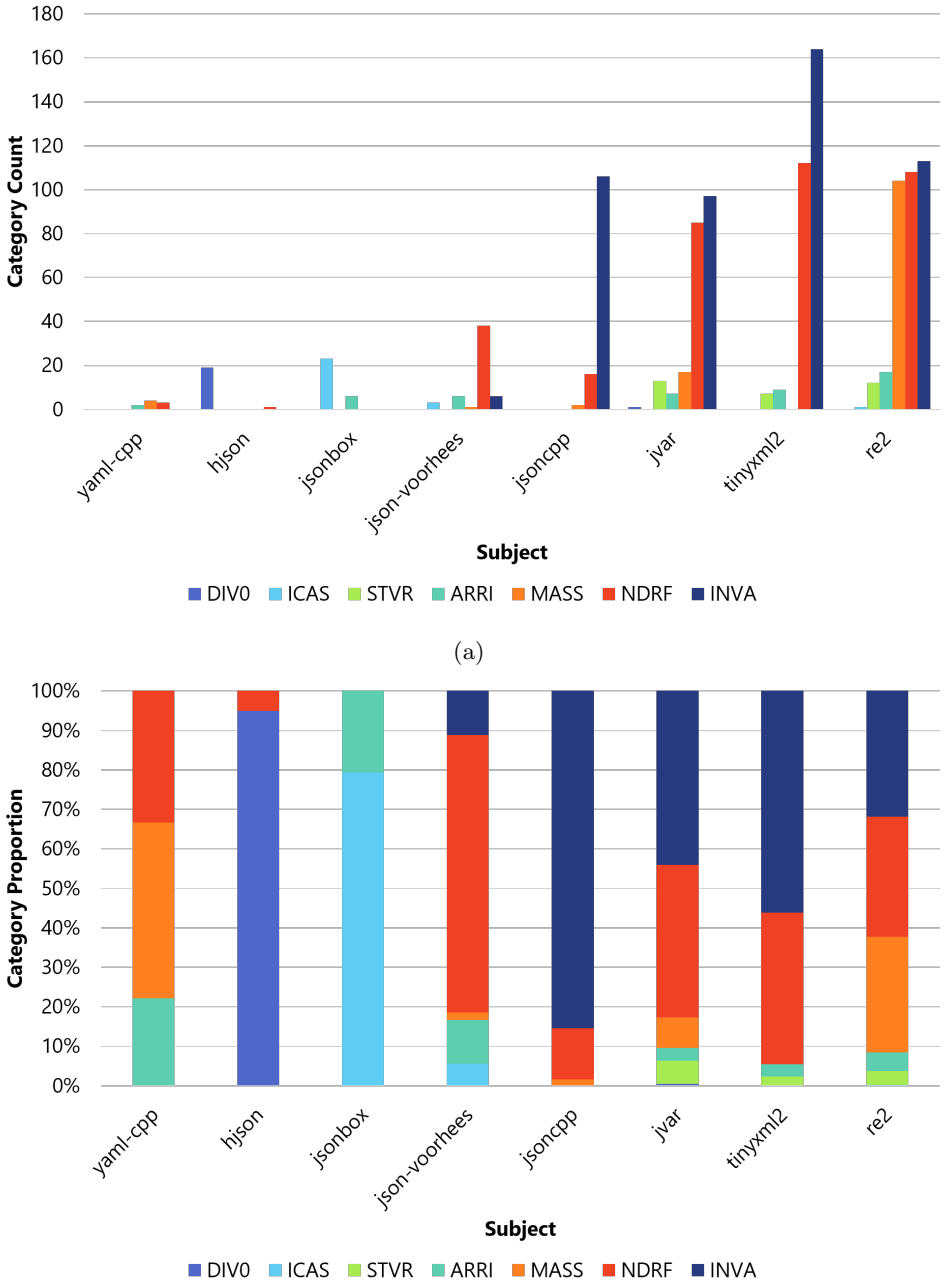

Figure 3.5: Category Distribution of False Positives per Subject

an unknown address). When CITRUS invoked `memberName` on `valueconstiterator1` at line A5, the program received a segmentation fault crash because the `valueconstiterator1`'s `current_` pointer (i.e., still pointing at an unknown address) was dereferenced (see "`(*current_)`" at line B2).

NULL pointer dereference (NDRF). The NDRF category occurs when the method call sequence generated by CITRUS performs address dereferencing on `NULL` pointers. There can be many ways the NDRF category happens in the method sequence generation by CITRUS. Some examples of frequently-occurring NDRF cases in the experiment were:

```
A1: re2::Prog prog12{};
A2: re2::Prog::Inst* inst13 = prog12.inst(-48);
A3: ...
A4: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A5: re2::EmptyOp emptyop70 = inst13->empty();
```

```
B1: Inst *inst(int id) { return &inst_[id]; }
```

Listing 3.2: Example of invalid address dereference (INVA) in `re2`

```
A1: Json::ValueIterator valueiterator0{};
A2: Json::ValueConstIterator valueconstiterator1{valueiterator0};
A3: ...
A4: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A5: const char* char22 = valueconstiterator1.memberName();
```

```
B1: char const* ValueIteratorBase::memberName() const {
B2:   const char* cname = (*current_).first.data();
B3:   return cname ? cname : "";
B4: }
```

Listing 3.3: Example of invalid use of iterator classes (INVA) in `jsoncpp`

1. CITRUS directly passed `NULL` argument on functions that were intolerant towards `NULL` values;
2. CITRUS performed dereference on pointers sourced from a return value of some functions that could return `NULL` values; and
3. CITRUS indirectly passed a structured variables (e.g., class instances) whose fields were not fully initialized (i.e., some fields were still `NULL`), and such uninitialized fields' value got dereferenced.

Similar to INVA category, the NDRF category was considered false positive because CITRUS failed to recognize the *runtime* value of pointers during generation. Crashes caused by dereferencing `NULL` values are indeed *trivial* and can not be argued to be a real bug. Listing 3.4 illustrates a crash example caused by the passing `NULL` to intolerant functions in `hjson`, as described below.

1. (Listing 3.4b) The function `Unmarshal` takes *two* arguments: (1) a `const char*`-typed pointer "`data`" at B2; and (2) "`dataSize`" that represents the array length of `data` at B3.
2. (Listing 3.4a) At A3, CITRUS passed `nullptr` as `data` argument to invoke `Unmarshal`.
3. (Listing 3.4b) At B6–B12, `Unmarshal` constructs a `Parser` object by passing both `data` and `dataSize` (B7–B8) as its constructor argument.
4. (Listing 3.4b) At B16, `Unmarshall` calls `_resetAt`, which perform a chaining call to `_next` (both definitions are shown in C1–C13).
5. (Listing 3.4c) The test case crashed because the expression "`p->data[p->indexNext++]`" at C9 led into iterating over a `nullptr` value (i.e., `p->data`).

```
A1: struct Hjson::DecoderOptions decoderoptions2{false, true, false};
A2: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A3: Hjson::Value value3 = Hjson::Unmarshal(nullptr, 218, decoderoptions2);
```

```
B1: Value Unmarshal(
B2:   const char *data,
B3:   size_t dataSize,
B4:   const DecoderOptions& options
B5: ) {
B6:  Parser parser = {
B7:    (const unsigned char*) data,
B8:    dataSize,
B9:    0,
B10:   ' ',
B11:   options
B12:  };
B13:  if (parser.opt.whitespaceAsCmts) {
B14:    parser.opt.comments = true;
B15:  }
B16:  _resetAt(&parser);
B17:  return _rootValue(&parser);
B18: }
```

```
C1: static void _resetAt(Parser *p) {
C2:  p->indexNext = 0;
C3:  _next(p);
C4: }
C5:
C6: static bool _next(Parser *p) {
C7:  // get the next character.
C8:  if (p->indexNext < p->dataSize) {
C9:    p->ch = p->data[p->indexNext++];
C10:   return true;
C11:  }
C12:  ...
C13: }
```

Listing 3.4: Example of `nullptr` dereference (NDRF) in `hjson`

Method assertion (precondition) failure (MASS). The MASS category was found mostly in crashes induced by assertion failure. In most programming languages, developers could write "assertions" on methods to eagerly fail (i.e., crash) the method execution when some preconditions and/or expectations were unsatisfied.

Crashes from the MASS category are generally false positives in the unit-level testing. This is because the responsibility of valid test cases construction (i.e., sequences with no violation of any method precondition) should be owned by the testers. Listing 3.5 describes a false positive crash encountered in `yaml-cpp`. For this case, method `OnMapEnd` contains an assertion at B8 to check if top element of `m_stateStack` must be equal to `State::WaitingForKey`. To satisfy the condition, CITRUS must call `OnMapStart` (B1–B4) because it is the only way to push `State::WaitingForKey` element to `m_stateStack` (B3). However, the call sequences shown in line A1–A19 crashed because the precondition of `OnMapEnd` was not satisfied (i.e., assertion failed). Note that a similar precondition checking happened on `OnSequenceEnd` method calls (A11–A13); however, the execution did not crash at those lines because the precondition checks were satisfied (i.e., *three* `OnSequenceStart` calls at A3–A5 were placed before the *three* `OnSequenceEnd` calls at A11–A13).

Invalid array index / size (ARRI). The ARRI category relates to crashes by accessing array elements with invalid indexing (e.g., index > the actual array size, or negative index). Also, a crash was considered ARRI when CITRUS unintentionally provided a an incorrect array size that was larger than its actual array size.

The ARRI category was considered false positive because crashes from accessing out-of-bound in-

```
 A1: YAML::Emitter emitter24{};
 A2: YAML::EmitFromEvents emitfromevents25{emitter24};
 A3: emitfromevents25.OnSequenceStart(mark21, basicstring22, 124, value23);
 A4: emitfromevents25.OnSequenceStart(mark21, basicstring22, 124, value23);
 A5: emitfromevents25.OnSequenceStart(mark21, basicstring22, 124, value23);
 A6: double double29 = 0.546024;
 A7: YAML::ostream_wrapper ostreamwrapper30{};
 A8: ostreamwrapper30.set_comment();
 A9: YAML::Node node32 = YAML::convert<int>::encode(double29);
A10: YAML::Node node33 = YAML::Clone(node32);
A11: emitfromevents25.OnSequenceEnd();
A12: emitfromevents25.OnSequenceEnd();
A13: emitfromevents25.OnSequenceEnd();
A14: YAML::as_if<double, double> asif37{node33};
A15: bool bool38 = iteratorvalue9.IsSequence();
A16: bool bool39 = iteratorvalue9.IsSequence();
A17: bool bool40 = iteratorvalue9.remove<double>(0.412408);
A18: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A19: emitfromevents25.OnMapEnd();
```

```
 B1: void EmitFromEvents::OnMapStart(...) {
 B2:  ...
 B3:  m_stateStack.push(State::WaitingForKey);
 B4: }
 B5:
 B6: void EmitFromEvents::OnMapEnd() {
 B7:  m_emitter << EndMap;
 B8:  assert(m_stateStack.top() == State::WaitingForKey);
 B9:  m_stateStack.pop();
B10: }
```

Listing 3.5: Example of method assertion failure (MASS) in `yaml-cpp`

dexes are trivial to be claimed as a real crash bug. Listing 3.6 shows an example where CITRUS failed to provide the proper array size in `yaml-cpp` subject. At line A2, CITRUS passed an incorrect array size to the constructor of `YAML::Binary`. Then, at line A4, CITRUS performed a *self*-equality checking on `binary4` by invoking `operator==` using (also) `binary4` as argument. Since the provided array size at `binary4`'s construction exceeded the actual size (i.e., 10), the test case crashed within the `for` loop at line B7–B10.

Passing address of stack variable to method that takes ownership (STVR). The STVR category happens when CITRUS passed an address of stack variable to a function/method that takes ownership of the pointer-typed argument. On such case, a *double free* crash would occur because multiple `free` operations (on the stack variable) happened after these *two* following events: (1) the end of the `main` stack lifetime ; and (2) the end of the function/method (that took the pointer ownership) lifetime.

Similar to the MASS category, the STVR category was considered false positive because writing valid test case sequences (i.e., including passing the correct stack/heap variable references) should be

```
A1: unsigned char char3[10] = "l1yMG3cAf";
A2: YAML::Binary binary4{char3, 18446744073709551614};
A3: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A4: bool bool5 = binary4.operator==(binary4);
```

```
B1: bool operator==(const Binary &rhs) const {
B2:   const std::size_t s = size();
B3:   if (s != rhs.size())
B4:   return false;
B5:   const unsigned char *d1 = data();
B6:   const unsigned char *d2 = rhs.data();
B7:   for (std::size_t i = 0; i < s; i++) {
B8:     if (*d1++ != *d2++)
B9:     return false;
B10:  }
B11:  return true;
B12: }
```

Listing 3.6: Example of invalid array size (ARRI) in `yaml-cpp`

```
A1: re2::Prefilter::Op op0 = re2::Prefilter::Op::AND;
A2: re2::Prefilter prefilter1{op0};
A3: std::vector<re2::Prefilter*> vector2{&prefilter1, &prefilter1};
A4: prefilter1.set_subs(&vector2);
A5: prefilter1.set_subs(&vector2);
```

```
B1: // Set the children vector. Prefilter takes ownership of subs and
B2: // subs_ will be deleted when Prefilter is deleted.
B3: void set_subs(std::vector<Prefilter*>* subs) { subs_ = subs; }
B4:
B5: // Destroys a Prefilter.
B6: Prefilter::~Prefilter() {
B7:   if (subs_) {
B8:     for (size_t i = 0; i < subs_->size(); i++)
B9:     delete (*subs_)[i];
B10:    delete subs_;
B11:    subs_ = NULL;
B12:  }
B13: }
```

Listing 3.7: Example of passing address of stack variable (STVR) in `re2`

accommodated by the testers. Listing 3.7 illustrates an example of STVR at `re2` in more detail. In this case, the `re2` developers mentioned (in the comment section, see B1–B2) that `Prefilter::set_subs` takes over the ownership of the `subs` argument, and frees the `Prefilter`'s `subs_` field during destruction of `Prefilter` (see B6–B13). However, CITRUS passed an address of a stack variable `vector2` (A4–A5) to `prefilter1`, resulting a double free crash to occur at the end of the test case execution.

```
1: char char2[3] = "Lg";
2: jsonv::path_element pathelement3{char2};
3: std::vector<jsonv::path_element> vector4{pathelement3};
4: jsonv::path path5{vector4};
5: jsonv::formats_builder formatsbuilder7{};
6: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
7: jsonv::formats_builder formatsbuilder8 =
8: formatsbuilder7.check_references(
9:   (std::vector<jsonv::formats>&) vector4, ...);
```

Listing 3.8: Example of incorrect type casting (ICAS) in `jsonbox`

```
A1: Hjson::Value value2{35};
A2: ...
A3: Hjson::Value value7 = value2.operator%=(value2);
A4: /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A5: Hjson::Value value8 = value2.operator%=(value2);
```

```
B1: Value& Value::operator%=(const Value& b) {
B2:   if (prv->type != b.prv->type || prv->type != Type::Int64) {
B3:     throw type_mismatch(
B4:       "The values must be of the Int64 type for this operation.");
B5:   }
B6:   prv->i %= b.prv->i;
B7:   return *this;
B8: }
```

Listing 3.9: Example of division-by-zero (DIV0) in `hjson`

Invalid type casting on template specialization types (ICAS). The ICAS category occurs when CITRUS generates an invalid method call sequence that incorrectly passed an incorrect-typed variable (i.e., different type) to the required argument's type. The ICAS category in CITRUS occurred due to CITRUS's internal bug, where CITRUS had failed to recognize the correct type bindings of some *template specialization*-typed variables. Fortunately, such ICAS cases had only ∼2% (=27/1,103) occurrences, and were mostly on `jsonbox` only (see Figure 3.4).

In the experiment, ICAS category was considered false positive because crashes caused by performing type casting to a different type (than the original) did not correspond to any implementation bug in the target program. Listing 3.8 shows an example of invalid type casting happened in `jsonbox`. At L7–L9, CITRUS passed `vector4` (which is a `vector` of `jsonv::path_element`, see L3) as the argument of `check_references` (which requires a `vector` of `jsonv::formats`). The test case crashed because the execution failed to interpret the address of `vector4` as a `vector` of `jsonv::formats`.

Division by zero (DIV0). The DIV0 category relates to performing an arithmetic division with a *zero* denominator. In C/C++, performing zero divisions *trivially* raises the SIGFPE arithmetic exception signal.

The DIV0 category was considered to be false positive because there is no actionable to mitigate

crashes caused by division-by-zero operations (e.g., throwing another exception would also trigger another crash if the exception was unhandled by the caller function). Listing 3.9 describes a crashing test case example at`hjson` where division-by-zero occurred. At A3, the test case assigned the `prv->i` value of `value2` (which was initially 35 at A1) to be 0 (see line B6). Then, the execution crashed at line A5 due to division by zero (i.e., CITRUS had invoked `operator%=` on `value2`, which was already 0).

Chapter 4. Discussion

In this section, I will elaborate several interesting lessons learned based on this work. I will point out several suggestions on CITRUS for future improvements, that might also work as well in automated unit-level testing tools for other programming languages.

4.1 Lessons Learned

4.1.1 Complexities in C++ Unit-level Testing

Automated generation of unit-level test cases in C++ programs possesses some unique complexities that do not happen in other object-oriented programming language, such as Java. There are three notable major differences of testing in C++ from Java programs, which result in an increase in complexity for CITRUS (targets C++) apart from other automated testing tools for Java.

Linking configuration requirement

The current version of CITRUS must prompt the user to manually specify an additional *linking configuration* (i.e., consisting the object files (`.o`) location and the linking flags to use) for building the intermediate executable files during method sequence generation stage. Such linking configuration are mostly not necessary for tools targeting Java programs (e.g., EvoSuite [2]) because the executable files can be built on the same *classpath* where the target program's classes exist. Compared to testing Java programs, testing C++ programs is more complex because linking configuration is required to generate unit-level test cases. Providing an appropriate linking configuration is essential to obtain the best result (i.e., high coverage test cases) from the test case generation. Otherwise, inappropriate settings would significantly drop CITRUS's testing effectiveness (as CITRUS shall waste too many attempts on uncompilable codes).

Complexity in testing C++ template classes

While working on CITRUS, I also found that testing C++ template classes might be more difficult than Java's generic classes (although both concepts share major similarities in overall). Template classes in C++ are instantiated *on-demand* during compilation, which means that the template classes are only compiled when a certain code (e.g., CITRUS's intermediate driver) requires some specialization of the template class. This implies that the GCOV-instrumented object files (`.o`) in the target project directory *do not* record the coverage of the template class definition if the template class was not used anywhere in the target program. On the other hand, Java Generics are implemented by applying *type erasure* [34] during bytecode compilation, causing the class instances to still exist (and certainly measurable for coverage) without any type specialization event. Testing template classes in C++ programs is more complex than Java Generics because currently there is still no good solution to test such *unused* template classes in the program and CITRUS can only obtain *new* coverage information from the GCOV log files (`.gcno`) in the target project directory.

Unsafe memory model and C legacy codes

Since C++ was built on top C programming language, the pitfalls of C memory models are inherited to programs which were built using C++. Some "*memory unsafe*" codes from the legacy C programming language still exist even in modern C++ programs, including methods that implicitly takes ownership, `'\0'`-terminated C-string assumption, pointer aliasing and object lifetime issues, and so on. To improve CITRUS effectiveness, it is inevitable that CITRUS must also accommodate to such legacy codes (although CITRUS targets modern C++ programs). On the other hand, more recent programming languages (e.g., Rust) offer language safety to prevent such memory corruption bugs.

4.1.2 Analyzing Crashing Test Cases

Analyzing crashing test cases is indeed crucial to uncover new bugs from the testing experiment. In this subsection, I will elaborate some interesting observations while conducting the analysis of crashing test cases in CITRUS experiment.

Stack hashing might be ineffective to de-duplicate unit-level crashes

In the conducted CITRUS experiment, CITRUS produced 1,103 de-duplicated crash reports (combined from 10 runs) in total, all of which must be manually analyzed to uncover new bugs. This is still large number of reports to be analyzed manually, especially with the high false positive rate in automated unit-level testing context. Moreover, the automatic crash de-duplication through stack hashing still counted many similar crashes into separate crash reports due to different initial stack traces (e.g., the following call stacks are counted as different crashes by the current CITRUS crash de-duplication scheme: "$A() \rightarrow X() \rightarrow Y()$" and "$B() \rightarrow X() \rightarrow Y()$"). Another better crash de-duplication scheme would be helpful to reduce the manual efforts from analyzing the large number of reported crashing test cases.

Correspondence of crashes to real bugs in unit-level testing context

Compared to system-level testing where crashes triggered by system-level inputs are frequently corresponding to real bugs, crashes in unit-level testing does not always relate to (nor imply the existence) some bugs in the target program. Although I have put extensive effort to analyze and find interesting *candidate crash bugs* from the large number of crashing test cases, the majority of unit-level crashes were *false positive crashes* (as described in Section 3.3.2) when such reports were brought to the upstream developers. For example, the crash caused by multiple calls to `Decref` in `re2` was indeed trivial due to the semantically-invalid call sequence. Even there could be some interesting cases where *seemingly*-valid crashing call sequences were actually "out-of-scoped" to be handled by the *library API*-level; instead, such crash prevention must be controlled by the developers in the *application*-level (e.g., the infinite recursion issue of cyclic reference in `jsoncpp`). These examples suggest that the *crashing* heuristic itself is insufficient to easily discover new bug in automated unit-level testing.

4.2 Suggestions

There are several suggestions to improve CITRUS in the future, as listed in the following items.

Function selection heuristics

Currently, CITRUS performs method call sequence generation to produce interesting test cases through *purely random* selection of functions from the entire target program. However, this might not be a cost-effective approach to *neither* maximize the test case coverage *nor* the bug detection capability. This is because some functions are mostly differ in term of complexities; and thus, they require more testing time to be allocated. A suggestion to improve the future version of CITRUS would be to prioritize functions based on combination of some selective criteria, such as: (1) functions with high complexity (e.g., cyclomatic, branches); (2) averaged time cost for function invocation; (3) number of invocation attempts; (4) functions with recent updates; (5) *success*-to-*failure* ratio; and so on. Additionally, CITRUS can also select function that *explicitly* mutates object states to boost the exploration of producing diverse object states, rather than relying on random function selection scheme to generate interesting object construction sequence.

Parameterized test case generation

In RQ2, CITRUS variants that utilize `libfuzzer` (i.e., $\texttt{C}_{6+\texttt{LF1}}$, $\texttt{C}_{6+\texttt{LF3}}$, and $\texttt{C}_{12+\texttt{LF2}}$) achieved competitively higher test coverage even in significantly less time budget (i.e., up to 14.6 hours less) compared to the CITRUS's $\texttt{C}_{24}$ variant. This comparative result indicates that CITRUS suffered performance bottleneck during method sequence generation through a series of repetitive *compilation* and *linking* operations to build the intermediate executable files. Even worse, such series of operations would waste most of the time budget when CITRUS produces too many "built-failure" executable files (e.g., due to incorrect linking configuration). Such heavy overheads did not occur during the `libfuzzer` stage after the method sequence generation because it only requires *one* time building process to build the fuzzing executables. This allow the `libfuzzer` stage to utilize more time budget to explore more diverse program states and discover new branches.

Based on this observation, a suggestion to improve CITRUS can be made by introducing *parameterized test case generation*. Compared to the current structure of CITRUS test case, a parameterized test case takes an additional input file as the input argument of the method calls. The input file will also be mutated in the process to reduce the number of redundant compilation and linking operations during the method sequence generation. This way, CITRUS can effectively allocate more time budget to improve its coverage achievement.

Reducing false positive rate

There can be several ways to reduce the number of false positive crashes reported by CITRUS, according to the false positive crash types elaborated in Section 3.3.2. Several suggestions to improve CITRUS towards limiting false positive rate are listed as follows:

1. To reduce false positives from the division-by-zero (DIV0) category, CITRUS can utilize static analysis to distinguish functions that performs arithmetic division and modulo operations. Then, CITRUS should utilize such information to prevent any *zero* value to be passed as a method argument.

2. Similarly, false positives from the `NULL` pointer dereference (NDRF) category can be prevented by distinguishing functions that were intolerant towards `NULL` pointers. For example, a function $f(x)$ that performs `strlen` on x argument should not receive `NULL` pointer as x argument. Using such

information, method call sequence generation while resolving arguments for functions with pointer-type can be controlled, which is only passing `NULL` values to functions that explicitly handles `NULL` values (e.g., those with an explicit `NULL`-checking branch).

3. False positives from invalid array indexes and sizes (ARRI) category frequently happened in methods whose signature contains *both* of the following types: (1) a pointer type and (2) an "*unsigned*" integral type (e.g., `size_t`). For such functions, CITRUS can use smaller set of integer values to prevent crashes triggered by *out-of-bound* array index accesses. Also, CITRUS must also consider to put a `'\0'`-value right at the end of every `char*`-typed pointer/array to prevent false crashes caused by C-string function calls (e.g., `strlen`) that assumes every `char*`-typed array are always `'\0'`-terminated.

4. To reduce the false positive crashes by object ownership issues (e.g., STVR), CITRUS should perform data analysis on every function with pointer-typed arguments, which implicitly takes the object ownership during invocation (e.g., a function $g(x)$ that frees the object referenced by x at the end of g call takes the ownership of x). For such functions, CITRUS should pass heap object references (i.e., constructed using the "`new`" keyword) instead of passing the addresses of stack variables (i.e., using address-of (`&`) operator).

Based on the large number of false positive crash reports, another false positive reduction improvement on CITRUS can also be made by applying a better crash de-duplication scheme. The current stack hashing approach still produces large number of redundant crash reports due to the inflexibility of utilizing complete traces of the function call stack. Limiting the stack hashing to consider only several *top-levels* of call stack (instead of *full* stack traces) would group similar crashes into better groupings; and hence it reduces the efforts to analyze CITRUS crashing test cases. Alternatively, CITRUS can apply crash clustering [35] approach for better crash de-duplication than stack hashing.

Test case minimization to improve usability

Currently, CITRUS only focuses to generate random method call sequences (with arbitrary length) to maximize the testing coverage in the target program. However, the length of test case frequently matter towards the test case usability because smaller test cases offer better understandability and less running time. An improvement could be made to CITRUS in this context by performing *test case minimization* during method sequence generation in order to produce more compact test cases.

Chapter 5. Related Works

This section will elaborate several previous works that are highly-related to CITRUS consisting of the following items:

1. recent approaches on automated unit-level test case generation for C/C++ programs; and
2. previous works that are related to method call sequence generation for testing object-oriented programs.

5.1 C/C++ Unit-level Testing Tools

Automated unit test case generation for C/C++ is well-known to be a challenging task due to highly-expressive language features. For example, C memory model allows *pointer aliasing* [36], which is remain to be prominent research topic [37, 38] (until now) as there is no good solution to overcome such problem. Moreover, C++ introduces object-oriented features to the language that remarkably increase the complexity of automated testing tools, such as inheritance relationship, object polymorphism, template classes, STL classes, and so on. These are some main reasons that motivates the limited availability of automated testing tools to target C++ programs at present time.

Apart from the highly-expressive language features, automated unit test case generation for C++ programs is also challenging for its large search space. To achieve a high coverage in a target program, testing tools must be able to: (1) generate diverse test harnesses (a.k.a., drivers) to represent realistic contexts to test each unit function in the target program; and (2) generate the suitable inputs to increase the test coverage. Most state-of-the-art techniques, such as symbolic executions (e.g., CUTE [39], KLEE [25], DeepState [40]) and coverage-guided fuzzing (e.g., AFL++ [18], `libfuzzer` [24]) require manual efforts by the human testers on providing the unit-level test harnesses before starting the input generation. Such manual interventions are indeed costly and ineffective [41] for achieving high test coverage [1]. Meanwhile, in contrast to C++ unit testing frameworks (e.g., Google Test [42], CppUnit [43]) that do not automatically generate test cases (i.e., purposefully only for running unit test cases), CITRUS generates C++ unit tests automatically which can be later incorporated to be run on these C++ testing frameworks.

The later approaches on automated unit testing started to incorporate the harness generation process inside the testing process. KLOVER [20] automatically generates static drivers which is later incorporated with its own C++ symbolic execution engines to generate inputs. Before performing input generation using their symbolic execution engine, KLOVER generates symbolic test drivers for declaring a set of symbolic variables that will be used to invoke the target function. FSX [21] introduces incremental driver refinement and relevant input analysis to improve the effectiveness of static drivers in symbolic executions. However, utilization of such static drivers may have limitation in triggering some particular behaviors caused by the different ordering of method calls. Moreover, the source code implementation of KLOVER and FSX are also unavailable for the public to test on their C++ programs. Compared to

[1] Performing experiment to achieve an apple-to-apple comparison between CITRUS and an existing tool (such as DeepState and AFL) was difficult because most existing tools do not automatically generate test drivers (i.e., only mutates the input bytes), while CITRUS generates the test drivers end-to-end.

KLOVER and FSX, CITRUS generates test cases through the random method call sequence generation to execute diverse program behaviors. The randomness in the method call sequence generation helps CITRUS to diversify the object states produced from arbitrary sequence of method invocations, which is a clear CITRUS' advantage over the static drivers.

Moving forward from the static drivers in symbolic executions, we dive into more recent techniques that focuses on synthesizing fuzz drivers to achieve high test effectiveness in fuzzing. For example, FUDGE [22] and FuzzGen [23] synthesize fuzz drivers by scanning existing external library consumer to collect candidate entry functions of a library-under-test (i.e., both FUDGE and FuzzGen target for particular library testing purpose). FUDGE [22] collects the candidate entry functions by scanning for all places where the target library API were used in the specified external project. Then, FUDGE synthesizes an individual fuzz driver (for each API use location) by referring to the call sequences (called *snippet*) that were found in the external project. Similar to FUDGE, FuzzGen [23] synthesizes fuzz drivers by leveraging an *Abstract API Dependence Graph* (AADG) of existing external projects to construct valid API call sequences of library API invocation. Contrary to FUDGE where the code snippet were originally *sliced* from the consumer code, FuzzGen extracts the possible API sequences from the AADG to construct independently from the snippet. This way, fuzz drivers created by FuzzGen eliminates all irrelevant consumer codes, except only the target library API calls alongside with their *minimum* data dependency to construct all the required API arguments. Both FUDGE and FuzzGen rely heavily on the external project, based on which the fuzz drivers are synthesized from. This could be disadvantageous due to some reasons, such as:

1. When some of the library APIs were not utilized anywhere in the external project. On such case, FUDGE and FuzzGen will fail to produce fuzz drivers on those unused APIs.

2. Fuzz drivers sourced from code snippets in external projects may not thoroughly explore diverse object states caused by the huge combinations of arbitrary method call sequences (i.e., due to the large search space of possible call sequences that may not exist in the external project).

On the other hand, CITRUS works in a more flexible way as it independently performs random method call sequence generation, and does not require any external project to generate test cases. Also, CITRUS constructs arbitrary sequences by recognizing and combining multiple *object creator* call statements to construct objects, which is another major difference of CITRUS from those two aforementioned approaches.

The most recent work of fuzz driver synthesis is IntelliGen [41] in 2021. IntelliGen does not rely on external library consumer, but synthesizes fuzz drivers for library APIs by prioritizing functions with the most potential vulnerable statements (e.g., pointer dereferencing). In other words, IntelliGen prioritizes such vulnerable functions to maximize the bug detection capability in the target library. However, I could not check if IntelliGen supports C++ language features since their implementation is not publicly available. Meanwhile, CITRUS was developed specifically to target complex C++ programs, and CITRUS is made to be publicly available.

5.2 Method Call Sequence Generation

Method call sequence-based test case generation has been widely applied to various programming languages other than C++, such as Java [1, 2] and Python [19]. In Java unit-level testing case, EvoSuite [2] is the current state-of-the-art tool based on search-based software testing (SBST). The core process

of EvoSuite is based on random method call sequence generation of object-constructing statements to construct test cases (similar to CITRUS). From these constructed individual test cases, EvoSuite then composes high-quality test suite (in term of coverage achievement and test case simplicity) through genetic evolutionary algorithm to maximize the coverage goals (e.g., line, branch, weak mutation) while still minimizing the test suite size. EvoSuite performs test suite minimization in both *test suite*-level (i.e., eliminating redundant test cases) and *test case*-level (i.e., eliminating redundant statements); both with the same ultimate goal which is to improve the test suite readability and reduce the risk of generating flaky tests [44]. To reach the current state of maturity level, EvoSuite has gone through many engineering efforts to improve its effectiveness and efficiency on testing modern Java programs, such as functional mocking [45], integration with virtual file system (VFS) [46], dynamic symbolic execution [47], multiple seeding strategies [48], and many more. Also, there has been several works [49, 50, 51, 52] on evolutionary search algorithms to improve the utilization of EvoSuite's search time budget more effectively.

Randoop [1] is another unit-level test generation tool targeting Java programs based on random testing. Randoop performs random method call sequence generation while utilizing *contract checkers* (i.e., methods to assert the invariant of a class) to find any violations within the code executions. Through these assertions, Randoop distinguishes a contract-violating sequences from the non-violating test cases. Then, for each non-violating test case, Randoop utilizes feedback from *filters* to store only the sequence that produces an interesting object state (i.e., has not been encountered previously). This is done to increase the diversity of object states by limiting Randoop to test on call sequences that produces the redundantly-created objects with same values. Compared to CITRUS (which is also a tool based on random method call sequence generation), there are two major differences between Randoop and CITRUS as follows:

1. To predict an unexpected behavior of the target program, Randoop relies on contract checking methods to discover contract-violating sequences; while CITRUS utilizes crashes in the program execution. Note that the availability of such contract checking assertions (required by Randoop) is almost none in most real-world C++ programs, which makes it inapplicable for fully-automated testing in CITRUS case.

2. To guide the method call sequence generation process, Randoop stores call sequences that produces a previously-unseen object state (i.e., checked through Java's object checking equality); while CITRUS uses test coverage as its guidance (i.e., only test cases that increase the coverage will be kept). Note that the object checking equality functions in C++ programs are less common than in Java, which makes it less applicable in C++ testing context.

Meanwhile, Garg et al. [53] has ported an implementation of Randoop's directed random testing to target C++ programming language. In their work, Garg et al. proposed a hybrid approach of combining Randoop's directed random testing with concolic execution to overcome coverage plateaus while testing C++ programs. Unfortunately, the implementation of their tool is not publicly available at present time.

Bach et al. [26] performed an empirical study to systematically construct valid object-constructing call statements in the context of object creation problem (OCP) while writing unit tests for C++ programs. To confirm the relevance of OCP, first they conducted a qualitative study on 143 developers to characterize their preferences while selecting an appropriate object creation code sequence for unit testing, particularly when there were *multiple* object creators for a particular class. From this study, they found that most developers tend to select the most non-sophisticated constructors (i.e., with the minimum dependencies) while constructing object creation code sequences for unit testing. As to produce

the *smallest* valid method call sequences, Bach et al. also proposed an algorithm (for constructing a particular class type) by traversing the shortest path on a call-dependency graph traversal. The evaluation showed that their approach was capable to construct *at least* a valid argument-constructing call sequences to invoke at least 94% functions on seven large C++ projects. While such approach sounds robust to construct *at least* a valid object-constructing call sequence for most classes, their work contains no discussion on how such method call sequences should be utilized to increase the diversity of object states for unit-level testing. The diversity of generated object states is essential because invoking functions using homogeneous objects could not enhance the testing effectiveness in automated test case generation context. Meanwhile, CITRUS constructs its object-constructing call sequences by utilizing *object creator* functions, which is similar to the approach mentioned above. However, compared to the approach, CITRUS is also interested in constructing diverse object states by performing random method call sequences generation and test case mutations.

Chapter 6. Concluding Remark

6.1 Conclusion

This paper presents CITRUS, which is a new automated C++ unit-level testing tool to generate random method call sequences to produce test suites achieving high test coverage. CITRUS is currently one of the very few available tools that automatically generate unit tests for C++ programs without human intervention. To test the highly-complex real-world C++ programs, CITRUS handles challenging technical issues of C++ language features, such as template instantiation, complex STL classes, and so on. To intensify the unique program behaviors discovered during the method call sequence generation, CITRUS applies `libfuzzer` fuzzing on discovered test cases to continue traversing *smaller* branches and improve the test coverage in significantly *less* testing time budget.

On eight real-world C++ target programs, I have demonstrated that CITRUS achieved high statement coverage (up to 95%) and high branch coverage (up to 79%) in the conducted experiment. I also have demonstrated that by applying `libfuzzer` fuzzing after the method sequence generation, CITRUS produced as competitively high testing capability with smaller time budget (saving up to 14.6 hours in average). Additionally, I have reported *seven* types of false positive crash alarm that frequently happen in C++ unit-level testing (with provided examples). To demonstrate CITRUS's bug detection capability, I have conducted a case study where CITRUS had successfully detected real crash bugs in the past versions of C++ programs.

For the continuous support on future researches of C++ unit-level automated testing tools, CITRUS is made available to public at

https://github.com/swtv-kaist/CITRUS

6.2 Future Works

As for future works, I plan to enhance the current CITRUS implementation by adding additional supports towards more complex C++ features, such as STL classes from the more recent C++ language features, function pointers, STL's threads and synchronization locks, and so on. Furthermore, in order to improve the testing effectiveness and efficiency of CITRUS, I plan to introduce a new function selection heuristic for CITRUS to prioritize complex functions that require more attention (i.e., to be extensively tested) than the simple functions, such as *getter* and *setter* methods. Also, I plan to apply parameterized test case generation during method sequence generation stage to save the testing time budget from redundant compilation and linking operations. Based on the false positive crash types I found in this work, I will also try to limit the number of false positive alarms produced by CITRUS through a more effective crash de-duplication scheme and with a more careful false positive filtering. Finally, I will study a method to identify/generate test oracles in C++ tests since test oracle problem is also an important problem for automated unit-level testing.

Bibliography

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," Proceedings of the 29th International Conference on Software Engineering, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2007.

[2] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, (New York, NY, USA), pp. 416–419, ACM, 2011.

[3] A. Arcuri, "RESTful API Automated Test Case Generation," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 9–20, 2017.

[4] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[5] I. S. W. B. Prasetya, "Random Testing with Austere Budgeting in T3: Benchmarking at SBST2019 Testing Tool Contest," in *Proceedings of the 12th International Workshop on Search-Based Software Testing*, SBST '19, p. 21–24, IEEE Press, 2019.

[6] M. Zalewski, "American Fuzzy Lop (AFL) Fuzzer." http://lcamtuf.coredump.cx/afl/, 2017.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 1032–1043, Association for Computing Machinery, 2016.

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

[9] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "SUSHI: A Test Generator for Programs with Complex Structured Inputs," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 21–24, 2018.

[10] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, "Automated Unit Testing of Large Industrial Embedded Software Using Concolic Testing," Automated Software Engineering (ASE), pp. 519–528, 2013.

[11] J. Ribeiro, "Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming," pp. 1819–1822, 01 2008.

[12] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 59–68, 2006.

[13] L. Li, Q. Dong, D. Liu, and L. Zhu, "The Application of Fuzzing in Web Software Security Vulnerabilities Test," 2013 International Conference on Information Technology and Applications, pp. 130–133, Nov 2013.

[14] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," vol. 00 of *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 660–677, 2018.

[15] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-Guided Property-Based Testing in Java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 398–401, Association for Computing Machinery, 2019.

[16] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, 2017.

[17] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, Sept. 2015.

[18] A. Fioraldi, D. Maier, H. Eißfeldt, , and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *In 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, Aug. 2020.

[19] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated Unit Test Generation for Python," in *Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8)*, vol. 12420 of *Lecture Notes in Computer Science*, pp. 9–24, Springer, 2020.

[20] G. Li, I. Ghosh, and S. P. Rajan, ""KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs"," in *"Computer Aided Verification"* (G. Gopalakrishnan and S. Qadeer, eds.), (Berlin, Heidelberg), pp. 609–615, Springer Berlin Heidelberg, 2011.

[21] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara, "FSX: Fine-Grained Incremental Unit Test Generation for C/C++ Programs," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 106–117, Association for Computing Machinery, 2016.

[22] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "FUDGE: Fuzz Driver Generation at Scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[23] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic Fuzzer Generation," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2271–2287, USENIX Association, Aug. 2020.

[24] "Libfuzzer – A Library for Coverage-guided Fuzz Testing." https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-09-28.

[25] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209–224, 2008.

[26] T. Bach, R. Pannemans, and A. Andrzejak, "Determining Method-Call Sequences for Object Creation in C++," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 108–119, 2020.

[27] "Default Constructors - cppreference.com." https://en.cppreference.com/w/cpp/language/default_constructor. Accessed: 2021-11-17.

[28] "Struct and Union Initialization - cppreference.com." https://en.cppreference.com/w/c/language/struct_initialization. Accessed: 2021-11-17.

[29] H. Agrawal, R. Demillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, and E. Spafford, "Design Of Mutant Operators For The C Programming Language," 10 1999.

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

[31] "LCOV Modified by henry2cox." https://github.com/henry2cox/lcov/tree/diffcov_initial. Accessed: 2021-10-01.

[32] "Initializer List - cppreference.com." https://en.cppreference.com/w/cpp/utility/initializer_list. Accessed: 2021-11-17.

[33] "Type Alias - cppreference.com." https://en.cppreference.com/w/cpp/language/type_alias. Accessed: 2021-11-17.

[34] "Type Erasure - Oracle's Java Documentation." https://docs.oracle.com/javase/tutorial/java/generics/erasure.html. Accessed: 2021-11-17.

[35] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash Deduplication Through Root-Cause Clustering," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, (New York, NY, USA), p. 3318–3336, Association for Computing Machinery, 2021.

[36] M. Emami, R. Ghiya, and L. J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, (New York, NY, USA), p. 242–256, Association for Computing Machinery, 1994.

[37] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 49–64, USENIX Association, Aug. 2015.

[38] D. Trabish, T. Kapus, N. Rinetzky, and C. Cadar, "Past-Sensitive Pointer Analysis for Symbolic Execution," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), p. 197–208, Association for Computing Machinery, 2020.

[39] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE), pp. 263–272, 2005.

[40] P. Goodman and A. Groce, "DeepState: Symbolic Unit Testing for C and C++," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[41] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "IntelliGen: Automatic Driver Synthesis for Fuzz Testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 318–327, 2021.

[42] "GoogleTest User's Guide." https://google.github.io/googletest/. Accessed: 2021-12-23.

[43] "CppUnit Test Framework." https://freedesktop.org/wiki/Software/cppunit/. Accessed: 2021-12-23.

[44] J. Campos, A. Panichella, and G. Fraser, "EvoSuite at the SBST 2019 Tool Competition," in *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pp. 29–32, 2019.

[45] A. Arcuri, G. Fraser, and R. Just, "Private API Access and Functional Mocking in Automated Unit Test Generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 126–137, 2017.

[46] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated Unit Test Generation for Classes with Environment Dependencies," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), p. 79–90, Association for Computing Machinery, 2014.

[47] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving Search-based Test Suite Generation with Dynamic Symbolic Execution," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 360–369, 2013.

[48] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding Strategies in Search-based Unit Test Generation," *Software Testing, Verification and Reliability*, vol. 26, pp. n/a–n/a, 03 2016.

[49] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[50] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating Branch Coverage as a Many-Objective Optimization Problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, 2015.

[51] A. Panichella, F. M. Kifetew, and P. Tonella, "A Large Scale Empirical Comparison of State-of-the-art Search-based Test Case Generators," *Information and Software Technology*, vol. 104, 08 2018.

[52] A. Arcuri, "Many Independent Objective (MIO) Algorithm for Test Suite Generation," pp. 3–17, 08 2017.

[53] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution," Proceedings of the 2013 International Conference on Software Engineering, (Piscataway, NJ, USA), pp. 132–141, IEEE Press, 2013.

Acknowledgment

First of all, I would give all praises and glory to Lord Jesus, for His everlasting love to me through all my joy and hardships in KAIST. I am really grateful to Him for all skillset He had entrusted me, and for the good health amid the COVID-19 pandemic while pursuing this Master degree.

I would like to show my sincere gratitude towards my dearest advising professor, Prof. Moonzoo Kim, for his care and gentleness towards me during my research life. His kindness and patience towards all SWTV Lab members have been very admirable to me. I am also admired for his strong motivation on teaching to always strive for the best, and I will be forever indebted for the many life-lessons he has taught me within the last two years. Therefore, I am more than thankful for this good opportunity to study at the SWTV Lab under his supervision.

I would like to express my deepest thanks to Prof. Yunho Kim and Prof. Shin Hong, for all the time they have invested in teaching me how to properly conduct research. I enjoyed having every technical discussion we had, and I really admire how well they communicate their ideas to make my research more useful. I would also like to thank the rest of the thesis committees, Prof. In-young Ko and Prof. Jongmoon Baik, for their insightful comments on this thesis.

Last but not least, I am really thankful to know all my fellow lab buddies: Ahcheong Lee, Zidong Yang, and Irfan Ariq; for being good friends during my stay at KAIST. I really hope that we could meet again one day to hear all your success stories in the future. I would also thank my dad, my mom, and my dearest sister who have made me accomplish this Master study and made me what I am today.

Curriculum Vitae

Name : Robert Sebastian Herlim

Birthplace : Surabaya, Indonesia

Educations

2020. 3. – 2021. 3. Korea Advanced Institute of Science and Technology (Master Program)

2014. 8. – 2018. 7. Bandung Institute of Technology (Bachelor Program)

2011. 7. – 2014. 7. Petra 2 Christian Senior High School Surabaya

Career

2018. 9. – 2020. 2. Full-stack Software Engineer at Airy Indonesia

Publications

1. **R. S. Herlim**, Y. Kim, and M. Kim, "CITRUS: Automated Unit Testing Tool for Real-world C++ Programs," *International Conference on Software Testing, Verification and Validation (ICST) Testing Tools track*, (forthcoming), 2022, IEEE.
2. **R. S. Herlim**, S. Hong, Y. Kim, and M. Kim, "Empirical Study of Effectiveness of EvoSuite on the SBST 2020 Tool Competition Benchmark," *International Symposium on Search Based Software Engineering*, 2021, pp. 121-135, Springer, Cham.
3. **R. S. Herlim** and A. Purwarianti, "Indonesian Shift-Reduce Constituency Parser Using Feature Templates & Beam Search Strategy," *2018 5th International Conference on Advanced Informatics: Concept Theory and Applications (ICAICTA)*, 2018, pp. 54-59, doi: 10.1109/ICAICTA.2018.8541292.