석 사 학 위 논 문
Master's Thesis

복잡한 C++ 프로그램을 위한 자동 유닛 테스팅 프레임워크

Automated Unit Testing Framework for Complex C++ Programs

2023

헤리얀토 이르판 아리크 (Heriyanto, Irfan Ariq)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

복잡한 C++ 프로그램을 위한 자동 유닛 테스팅 프레임워크

2023

헤리얀토 이르판 아리크

한 국 과 학 기 술 원

전산학부

복잡한 C++ 프로그램을 위한 자동 유닛 테스팅 프레임워크

헤리얀토 이르판 아리크

Automated Unit Testing Framework for Complex C++ Programs

Irfan Ariq Heriyanto

Advisor: Moonzoo Kim

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

Daejeon, Korea
June 13, 2023

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 헤리얀토이르판 아리크. 복잡한 C++ 프로그램을 위한 자동 유닛 테스팅 프레임워크. 전산학부 . 2023년. 58+iv 쪽. 지도교수: 김문주. (영문 논문)
Irfan Ariq Heriyanto. Automated Unit Testing Framework for Complex C++ Programs. School of Computing . 2023. 58+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

C++ 언어는 매우 많이 사용되는 언어임에도 불구하고, C++가 지원하는 매우 복잡한 기능때문에 C++ 프로그램을 테스트하는 것은 매우 큰 과제로 남아있다. 예를 들어, 템플렛, non-public 함수, 복잡한 STL 데이터 타입, 등의 복잡한 기능이 존재한다. 현재 이런 복잡한 기능을 다룰 수 있는 자동화된 유닛 테스트 도구는 전무한 실정이다.

본 논문에서는 이러한 복잡한 기능을 다룰 수 있는 새로운 자동화 유닛 테스트 생성 도구인 CLEMENTINE을 제시한다. CLEMENTINE은 기존 도구인 CITRUS의 3가지 제한점을 극복하고 더 많은 C++ 기능을 지원하도록 확장되었다. 그 결과, CLEMENTINE은 기존의 CITRUS가 테스트 생성에 실패한 4개의 실제 C++ 오픈 소스 프로그램을 효과적으로 테스트 할 수 있었다. 또한, 8개의 C++ 실제 오픈소스 프로그램에 실험한 결과, CLEMENTINE은 81.6%의 구문 커버리지, 60.1%의 분기 커버리지, 88.5%의 함수 커버리지를 보이며, 15.0%p 더 낮은 구문 커버리지, 8.9%p 더 낮은 분기 커버리지, 24.4%p 더 낮은 함수 커버리지를 보인 CITRUS에 비해 월등한 테스트 성능을 보이는 것을 확인했다.

핵 심 낱 말 자동화 테스트 생성, 무작위 함수 호출 시퀀스 생성, C++ 유닛 테스팅

Abstract

C++ is a very popular programming language. However, testing C++ programs is a challenging task due to the high complexity of C++ features (e.g., template, non-public member function, complex STL types, etc.), and there are almost no automated unit testing tool that handles such highly complex C++ features. I have developed CLEMENTINE, an automated unit testing tool for real-world C++ programs that handles complex C++ features. CLEMENTINE extends CITRUS by resolving three main limitations in CITRUS and supports more C++ features (e.g., testing non-public member function, handling global operator overloading, etc.). As a result, CLEMENTINE generates effective unit test cases for four real-world C++ programs that CITRUS fails to test. Moreover, the experiment results on eight real-world C++ open source programs show that CLEMENTINE could achieve 81.6% statement coverage (15.0%p higher than CITRUS), 60.1% branch coverage (8.9%p higher than CITRUS), and 88.5% function coverage (24.4%p higher than CITRUS) on average, proving that CLEMENTINE has better testing performance compared to CITRUS.

Keywords Automated test case generation, random method call sequence generation, C++ unit testing.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Background

1.1.1 Research Background

Software testing is one of the most important parts of software development. The main goal of software testing is to ensure that the programs meet the desired requirements and function correctly. Manual testing, while essential, can be a challenging and time-consuming task. It requires human testers to manually write and execute test cases that cover various scenarios. As software systems become more complex, the need for efficient and reliable test case generation techniqe has become increasingly important.

In response to the challenges of manual testing, the research community has devoted considerable effort to developing automatic software testing techniques to generate test cases. Various automated software testing techniques have been developed for the past decade. Blackbox random testing [2, 3], coverage-guided greybox fuzzing [4, 5, 6], symbolic execution [7, 8], and AI-driven search-based software testing (SBST) [9, 10] are some software testing techniques that have been developed. Furthermore, the usage of automated software testing has been proved for its practical usage like identifying security vulnerabilities in real-world systems[11, 12, 5, 13, 14]. Additionally, a recent study reported that automated software testing achieve produces test cases with higher test coverage compared to manually-written test cases [15].

However, many automated software testing techniques primarily focus on system-level testing [4, 5, 6, 16], which tests the software as a whole in its operational environment. While system-level testing is important for evaluating the overall behavior and integration of components, it may struggle to uncover vulnerabilities deeply embedded in the target code System-level testing often relies on high-level inputs and configurations, which may not adequately exercise all code paths or corner cases. For example, complex input combinations, subtle coding errors, or specific corner cases may go unnoticed during system-level testing. Additionally, system-level testing typically operates at a higher level of abstraction, making it difficult to isolate and pinpoint specific issues in individual units or components of the software.

This limitation of system-level testing can be overcome by unit-level testing. Unit-level testing enables developers to test individual units or components of the software in isolation, allowing for precise identification of defects and targeted testing There have been some recent works on automated software testing techniques specifically designed for unit-level testing [2, 10, 8, 17]. Unfortunately, there is not much work on automated software testing for C++ programs due to highly-complex C++ features.

This thesis focuses on automated software testing in unit-level test case generation for C++ programs. C++ stands as one of the most popular programming languages, renowned for its numerous positive aspects that contribute to its widespread usage. One notable advantage is its high performance, making it an ideal choice for resource-intensive applications. For instance, web browsers like Mozilla Firefox and Google Chrome and game engines like Unreal Engine and Unity, which power many popular games, are built using C++ to harness its performance capabilities. However, those complex C++ features (e.g., STL library, template instantiation, member accessibility, etc.) also make testing C++ programs more challenging and the availability of automated testing tools in unit-level for C++ is low.

1.1.2 Previous Approach

Unit-level testing plays a crucial role in program development. Automated unit-level testing tool is an important tool that generates test cases automatically. Randoop [2] and EvoSuite [10] are some examples of automated unit-level testing tool for Java programs. Pynguin [17] is an automated unit-level testing tool for Python programs. Unfortunately, there are only a few automated unit-level testing for C++ programs due to complex C++ features.

Coverage-guided greybox fuzzing (e.g., AFL++ [16], `libfuzzer` [18], POWER [19]) and symbolic executions (e.g., CUTE [20], KLEE [21], DeepState [22]) are automated testing technique for C++ programs that run in system-level. Coverage-guided greybox fuzzing generates various test cases by performing bytes mutation. Symbolic execution generates various test cases that can explore diverse paths in the target program by solving the symbolic path formula using an SMT solver. Although those techniques can be used to test C++ in unit-level, there is a need to write a test driver for each function.

Recent work tries to mitigate the high cost of writing test drivers by automatically generating test drivers for each function [23, 24, 25, 26, 27, 28]. FUDGE [25] and FuzzGen [26] automatically generate a test driver for each function used in the target library consumer by utilizing the consumer code. These works heavily rely on the consumer code and may fail to generate test drivers for rarely used functions in the target library. Moreover, these works focus on testing C++ libraries and are not suitable for testing an independent program. Meanwhile, UTBotCPP [28] generates a fuzz driver for each function in the target program and then uses a symbolic execution tool, KLEE [21], to generate various test input. Unfortunately, UTBotCPP still does not support many C++ complex features such as template and C++ standard library.

1.2 Thesis Statement and Contribution

1.2.1 Thesis Statement

The thesis statement for this work is written as follows:

Automated C++ unit testing can improve its test coverage for a broad range of target programs by handling *complex object-oriented features in C++ programming languages.*

Thus, the core section in this thesis is Section 2.3

1.2.2 Contributions

The contributions of this thesis are as follows:

1. I have classified 20 types of C++ functions that are not properly handled by CITRUS [1] (an automated C++ unit testing framework developed by KAIST SWTV group) due to the complex C++ features and developed solutions for 16 of them (see Section 2.3 for the detail).

2. I have improved the applicability of an automated C++ unit testing framework by addressing the three limitations in the design choices of CITRUS. As a result, CLEMENTINE successfully gener-

ates valid test cases on the four new target subjects while CITRUS failed to do it (see Section 2.2 for the detail).

3. I have performed experiments to empirically demonstrate the advantage of CLEMENTINE on the 16 C++ target subjects. CLEMENTINE achieved 34.9% to 96.6% function coverage (75.4% on average), 5.5% to 77.6% branch coverage (41% on average), and 11.9% to 95.3% statement coverage (62.0% on average) (see Section 3.2 for the detail).

1.3 Structure of Thesis

The remainder of this thesis is structured as follows: Chapter 2 explains the CLEMENTINE framework in detail. Chapter 3 describes the setup of the experiment and reports the experiment results to show the testing performance of CLEMENTINE. Chapter 4 lists related work to automated software testing for C++ programs and automated software testing that uses method call sequence generation technique. Lastly, Chapter 5 concludes this thesis and lists future work.

Chapter 2. CLEMENTINE Framework

2.1 Limitation of CITRUS

2.1.1 Improper Way of Writing Test Case Files

To call a function in the target subject, the test case is required to have at least the declaration of the function. CITRUS includes all the header files in the test case in order to put as many function declarations as possible in the test case. Although including all the header files in the test case may work in some cases, it is not a good approach since it can cause uncompilable problems.

Figure 2.1 shows the example of an uncompilable error caused by including all header files in the test case. Function `toStringV` is declared in header file `stringutil.hpp` at line A4 and is defined in `stringutil_impl`.hpp header file at line B2. Then, file `stringutil_impl.hpp` is included in file `stringutil.hpp` at line A8 and this makes file `stringutil.hpp` also contain the definition of function `toStringV`. Finally, CITRUS includes all header files in the test case (i.e., `test_case.cpp`) including `stringutil.hpp` and `stringutil_impl.hpp`. Thus, the test case contains two definitions of function `toStringV` (i.e., one definition from `stringutil.hpp` and another one from `stringutil_impl.hpp`) and it causes uncompilable error (i.e., `redefinition of 'toStringV'`). This is a severe problem since not being able to compile the test case means the test case cannot be executed and become useless.

Additionally, including all header files in the test case limits the number of functions that can be called in the test case. Consider the example code shown in Figure 2.2. There are 5 functions declared in the header file (i.e., at lines A1, A4, A5, A6, and A9) and 4 functions not declared in the header file (i.e., at lines B2, B4, B8, and B9). Just including the header file in the test case limits the number of functions that can be called in the test case to 5 functions that are declared in the header file. Thus, we miss the opportunity to test the other 4 functions that are not declared in the header file. Subsection 2.2.4 explains how this problem is solved in CLEMENTINE.

2.1.2 Creating Executable File of The Test case

Creating an executable of the generated test case is one of the important processes during testing. This process consists of 2 steps (i.e., compiling and linking). Without the correct compile command and link command, the compile process and the linking process can fail. Failure to create the executable from the generated test case will make the generated test case become useless since it cannot be run.

CITRUS utilizes a compilation database to get the correct compile command. The compilation database contains the compilation flags used to preprocess and compile the target file and it is emitted by C++ build tools like CMake [29] and BEAR [30]. Figure 2.3 shows the example of a compilation database from `clip`

However, such compilation databases do not provide information about the linking configuration (e.g., necessary object files to build an executable file). Thus, CITRUS needs its user to manually specify the linking configuration. The linking configuration consists of (1) the build directory location where the object files of the target subject are created, and (2) linking flags to additional external

```
// file: stringutil.hpp
A1: class StringUtil {
A2:   public:
A3:     template <typename... T>
A4:     static std::vector<std::string> toStringV(T... values);
A5:   ...
A6: }
A7: ...
A8: #include "stringutil_impl.hpp"
```

```
// file: stringutil_impl.hpp
B1: template <typename... T>
B2: std::vector<std::string> StringUtil::toStringV(T... values) {
B3:   ...
B4: }
B5: ...
```

```
// file: test_case.cpp
C1: #include "stringutil.hpp"
C2: #include "stringutil_impl.hpp"
C3: #include "..." // include other header files
C4: int main () {
C5:   ...
C6:   return 0;
C7: }
```

```
// Uncompilable Error Message
error: redefinition of 'toStringV'
std::vector<std::string> StringUtil::toStringV(T... values) {
                                     ^
...
```

Figure 2.1: Example of uncompilable error in `clip`

libraries (if any, e.g. `-lfmt` to use the fmtlib library and `-lz` to use zlib library). Providing incorrect linking configurations would degrade the testing effectiveness since the test case cannot be executed and become useless. Therefore, specifying the correct configuration is very important. Subsection 2.2.4 explains how CLEMENTINE solves this problem.

2.1.3 Many Non-Properly-Handled Functions

The third limitation of CITRUS is it has many non-properly-handled functions due to complex C++ features. Failure to properly handle function may result in generating uncompilable test cases or unlinkable test cases and hinder the testing performance. Thus, it is important to properly handle various types of functions. Section 2.3 explains 20 non-properly-handled function types and how CLEMENTINE solves them.

```
// file: header.hpp
 A1: void f();                // global function
 A2: class PublicClass {
 A3:  public:
 A4:   PublicClass(int x);    // constructor
 A5:   void setValue(int x);  // public member function
 A6:   int getValue();        // public member function
 A7:  private:
 A8:   int value
 A9:   void incrementValue(); // non-public member function
A10:   ...
A11: };
```

```
// file: implementation.cpp
 B1: #include "header.hpp"
 B2: static void h() {...} // static function
 B3: namespace {
 B4:  void func_anonnamespace() {...} // function inside anonymous namespace
 B5: }
 B6: class internal { // class that not declared in header files
 B7:  public:
 B8:   void func1() {...} // member function of class not declared in header files
 B9:   void func2() {...} // member function of class not declared in header files
B10: }
B11: ... // definition of functions declared in the header file
```

Figure 2.2: Example of functions not targeted by CITRUS

2.2 Architecture of CLEMENTINE

2.2.1 Overview of CLEMENTINE

CLEMENTINE is an automated unit-level testing tool based that generates test drivers using random method call sequence generation. CLEMENTINE the next version of CITRUS [31, 1] and it was developed by addressing limitations explained in section 2 and supporting more complex C++ features. Figure 2.4 [1] illustrates the overview CLEMENTINE's process.

2.2.2 Definition of CLEMENTINE's Test Case

A test case generated by CLEMENTINE consists of a sequence of method call statements and statements to instantiate objects for the method call's arguments. The test case does not have branching statements (e.g.,`if` statements). There are only two types of statements. Those two types are:

1. Primitive type variable declaration and initialization. For example, the statement "`bool bool1 = false;`" declares a variable named `bool1` with the primitive type (i.e., `bool1`) and initializes its value with `false`.

[1]This figure is directly taken from CITRUS [1, 31].

```
[
 {
  "directory": "/.../clip/build_temp",
  "command": "/usr/bin/clang++  -I/usr/include/cairo -I/usr/include/freetype2
      -I/usr/include/harfbuzz -I/usr/include} -I/.../clip/src -I/.../clip/src/utils
      -I/.../clip -I/.../clip/build_temp -g -O0 -fsanitize=fuzzer-no-link
      --coverage --save-temps -fPIC -std=gnu++17 -o
      CMakeFiles/clip.dir/src/api.cc.o -c /.../clip/src/api.cc",
  "file": "/.../clip/src/api.cc"
 },
 {
  "directory": "/.../clip/build_temp",
  "command": "/usr/bin/clang++  -I/usr/include/cairo -I/usr/include/freetype2
      -I/usr/include/harfbuzz -I/usr/include} -I/.../clip/src -I/.../clip/src/utils
      -I/.../clip -I/.../clip/build_temp -g -O0 -fsanitize=fuzzer-no-link
      --coverage --save-temps -fPIC -std=gnu++17 -o
      CMakeFiles/clip.dir/src/arrows.cc.o -c /.../clip/src/arrows.cc",
  "file": "/.../clip/src/arrows.cc"
 },
 ...
]
```

Figure 2.3: Example of Compilation Database

2. Method invocation. For example, the statement "ClassA obja1;" invokes a constructor ClassA without any argument.

The test case generated by CLEMENTINE has the following characteristics:

1. Every statement has a type and the type modifier if any.
2. Every statement has a variable with a distinct name, except a method invocation statement that returns `void`.
3. The value of every variable never changed (i.e., assigned only once).

2.2.3 CLEMENTINE's Process

This section explains the process of test case generation in detail. There are three main steps in the CLEMENTINE's process. Those three main steps are as follows:

1. Pre-processing Phase
2. Test Case Generation
3. Post-Processing The Generated Test Cases

(1) Pre-processing Phase

There are four things to do in the pre-processing phase. Those are (1) creating program representation, (2) creating test case template, (3) getting compile command, and (4) getting link command.

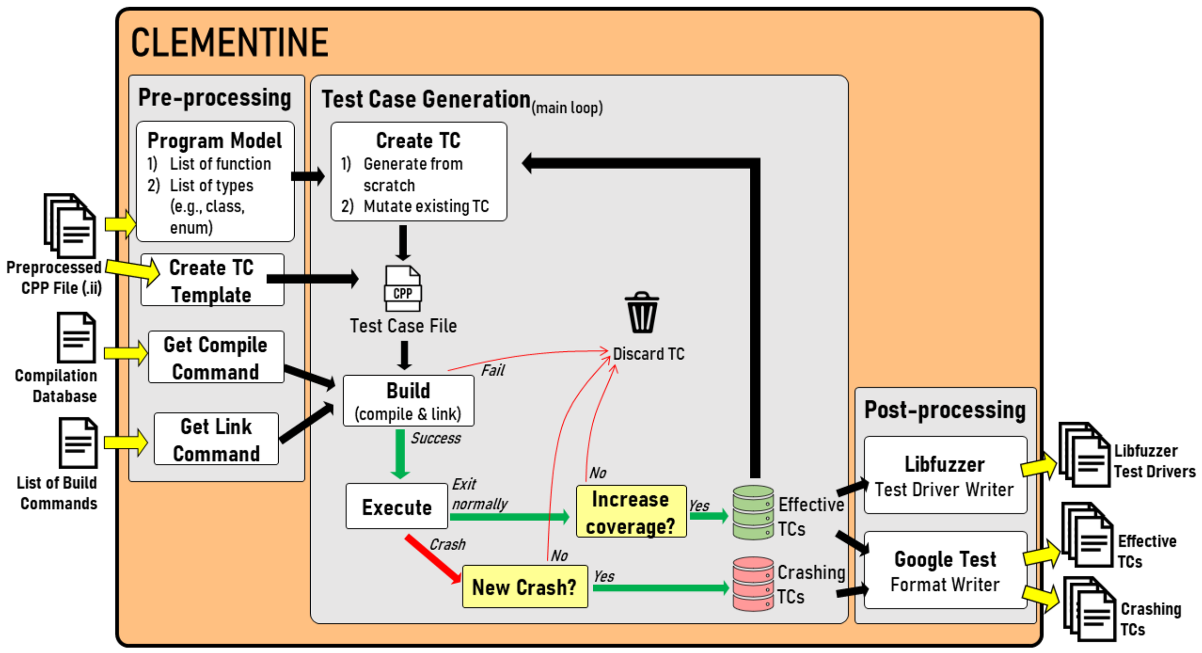


Figure 2.4: Overview of CLEMENTINE's process*

* This figure is directly taken from CITRUS [1, 31]

Creating Program Representation

To create program representation, CLEMENTINE collects the below information from the target program source code (i.e., preprocessed file `.ii`):

- Lists of `classes`, `structs`, `enums`, and global functions declared in the target program.
- Type used to specialize the templated function and class. This will be used in the template instantiation (subsection 2.2.4).

Next, CLEMENTINE creates a type system `TS` for `classes`, `structs`, `enums`, and functions in the target program. Algorithm 1 [2] describes how the type system `TS` of the target program is created. CLEMENTINE also creates an inheritance tree model (ITM) to utilize inheritance relationships for subclass instantiation (i.e., CLEMENTINE may construct an instance of the derived class to invoke a member function of the parent class).

CLEMENTINE differentiate functions into two types, "object creators" and regular functions. Several works on method sequence generation techniques [10, 32] also use a similar approach. In CLEMENTINE, Object creator is defined as a function that can provide an instance of a class type. Any function f that has *public accessibility* is considered an object creator of a class type X if the function satisfies one of the following conditions:

1. The function is a constructor of a class X (Note that, copy constructor, move constructor, and assignment operator (i.e., `operator=`) are not considered as object constructor)
2. The function is a global function and returns a non-primitive type X where $X \notin$ `ArgTypes`(f)
3. The function is a member function of a class Y and returns a non-primitive type X where $X \notin$ `ArgTypes`(f) and $X \neq Y$

[2]This algorithm is directly taken from [1, 31] since CLEMENTINE uses the same algorithm.

Algorithm 1: Creating Program Representation*

* This algorithm is directly taken from CITRUS [1, 31]

```
   Data: classes, enums, glob_fns from AST traversal
   Result: Inheritance tree model ITM and initialized type system TS
1  TS ← $\emptyset$; ITM ← $\emptyset$;
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.ExcludeNotTargettedFunctions();
17 repeat
18     TS.ExcludeUnsatisfiableFunctions();
19 until All fn in TS have satisfiable arguments;
```

4. The function is a **static** member function of a class Y and returns a non-primitive type X where $X \notin$ `ArgTypes`(f)

Figure 2.5 show an example of how CLEMENTINE recognizes a function as an object creator. There are 4 object creators in Figure 2.5. Those 4 object creators are:

1. function `FromCartesion` at line 3 (because it satisfies the third condition)
2. function `RandomPosition` at line 10 (because it satisfies the second condition)
3. function `getPoint1` at line 13 (because it satisfies the fourth condition)
4. implicit constructor of class `Line` (because it satisfies the first condition)

Function `setX` at line 4 is not recognized as an object creator of class `Point` because the return type of the function and its function owner is the same (i.e., class `Point`) and it is not a static member function. Function `mirror` at line 5 is not recognized as an object creator of class `Point` because X where `Point` $\in$ `ArgTypes`$(mirror)$. While the constructor of class `Point` declared in line 8 is not recognized as an object creator because its accessibility is not public.

CLEMENTINE filter out not targeted function and unsatisfiable function from the list of the function. CLEMENTINE defines a function as unsatisfiable if the function requires an instance of a type which CLEMENTINE cannot construct. For example, a function that requires an instance of a class

```
class Point {
 public:
  static Point FromCartesian(double x, double y) { return Point(x, y); } // object creator
  Point * setX(double _x) { x = _x; return this; } // NOT an object creator
  static Point mirror(Point pt) { return FromCartesian(pt.y, pt.x); }  // NOT an object creator
  double x, y;
 private:
  Point(int _x, int _y) { x = _x; y = _y; } // NOT an object creator
};
Point RandomPosition(); // object creator
class Line { // HAS object creator
 public:
  Point * getPoint1() { return &pt1; }; // object creator
  Point pt1, pt2;
};
```

Figure 2.5: Example of Object Creators*

* This code example is directly taken from CITRUS [1, 31] and has been modified

type with no detected object creators. Also, CLEMENTINE intentionally does not target several types of functions, for example, the constructor of non-instantiable class, deleted function, etc. Section 2.3 explains some types of functions that are not targeted by CLEMENTINE and unsatisfiable functions.

Creating Test Case Template

CLEMENTINE creates a test case template by copying the content of the preprocessed input file. CLEMENTINE also added some "driver" functions in the test case template. The "driver" function is explained in the subsection 2.3.10. This test case template will be included in the test case file. Subsection 2.2.4 explains how CLEMENTINE uses the test case template.

Getting Compile Command

CLEMENTINE utilizes the compilation database generated by the target program's build tool to get the compile command. The compilation database is generated in JSON (i.e., JavaScript Object Notation) format. Figure 2.3 shows an example of a compilation database. It contains a list of files compiled in the target program, the compile command used to compile the file, and the directory where the compilation occurred. CLEMENTINE utilizes this information to construct compile command for the test case file.

Getting Link Command

CLEMENTINE utilizes the build commands used to build the target program to get the linking command for the test case file. Figure 2.6 shows an example of list build commands that are used to build `clip`. CLEMENTINE distinguishes the linking command from the list of build commands by recognizing the tool used in the command. So far, there are two tools that are considered as linking commands. Those two tools are `ar` (i.e., command to create, modify, and extract archives [33]) and c++ tool (e.g., `gcc`, `g++`, `clang`).

```
1: ...
2: make[2]: Entering directory '/.../clip/build_temp'
3: /usr/bin/clang++  -I/usr/include/cairo -I/usr/include/freetype2
   -I/usr/include/harfbuzz -I/usr/include} -I/.../clip/src -I/.../clip/src/utils
   -I/.../clip -I/.../clip/build_temp -g -O0 -fsanitize=fuzzer-no-link --coverage
   --save-temps -std=gnu++17 -MD -MT CMakeFiles/clip-cli.dir/src/cli.cc.o -MF
   CMakeFiles/clip-cli.dir/src/cli.cc.o.d -o CMakeFiles/clip-cli.dir/src/cli.cc.o
   -c /.../clip/src/cli.cc
4: /home/irfanariq/local_tool/bin/cmake -E cmake_link_script
   CMakeFiles/clip-cli.dir/link.txt --verbose=1
5: /usr/bin/clang++ -g -O0 -fsanitize=fuzzer-no-link --coverage --save-temps
   "CMakeFiles/clip-cli.dir/src/cli.cc.o" CMakeFiles/clip.dir/src/api.cc.o
   CMakeFiles/clip.dir/src/arrows.cc.o ...
   CMakeFiles/clip.dir/src/utils/stringutil.cc.o
   CMakeFiles/clip.dir/src/utils/wallclock.cc.o CMakeFiles/clip.dir/src/vmath.cc.o
   -o clip  /usr/lib/x86_64-linux-gnu/libcairo.so
   /usr/lib/x86_64-linux-gnu/libfreetype.so
   /usr/lib/x86_64-linux-gnu/libharfbuzz.so /usr/lib/x86_64-linux-gnu/libpng.so
   /usr/lib/x86_64-linux-gnu/libz.so /usr/lib/x86_64-linux-gnu/libfontconfig.so
   /usr/lib/libfmt.a
```

Figure 2.6: Example of a file containing build commands to build `clip`

(2) Test Case Generation

Algorithm 2 [3] shows the main loop of CLEMENTINE. By default, CLEMENTINE operates in two stages: (1) deterministic stage, and (2) random stage. In the deterministic stage, CLEMENTINE generates a test case for each function by iterating the list of functions. The deterministic stage will be done once CLEMENTINE is finished iterating the list of functions (see algorithm 3). The purpose of iterating the list of functions during the deterministic stage is so CLEMENTINE generates **at least one** test case for each function. CLEMENTINE also does not perform test case mutation during the deterministic stage.

CLEMENTINE generate a test case by executing function `LoadOrGenerateTestCase` at line 5. Next, CLEMENTINE performs test case mutation by calling `MutateTC` at line 7 if the deterministic stage is done. Then, CLEMENTINE builds the test case executable `exe` by compiling and linking the test case file. CLEMENTINE executes the executable `exe` if the compilation and linking are successful. The test case is considered as crashing a test case if the execution did no terminates normally. In such case, CLEMENTINE re-executes the test case using `gdb` to get the stack trace of the test case. If the obtained stack trace has never been seen before, CLEMENTINE will save the test case in Q_{crash}. However, if the execution terminates normally, CLEMENTINE will measure the coverage of the target program. CLEMENTINE will save the test case in $Q_{effective}$ if the coverage is increased, otherwise the test case will be discarded. CLEMENTINE will stop these processes if the given timeout is reached.

Algorithm 3 [4] (`LoadOrGenerateTestCase`) shows how CLEMENTINE creates a new test case. During the deterministic stage, CLEMENTINE iterates the list of functions and generates a test case from scratch for the target function (i.e., L2–L7). The random stage will start if CLEMENTINE finishes iterating

[3]This algorithm is taken from [1, 31] and modified since the algorithm of CLEMENTINE and CITRUS is similar
[4]This algorithm is taken from [1, 31] and modified in CLEMENTINE.

Algorithm 2: Main Loop*

* This algorithm is directly taken from CITRUS [1, 31]

```
Data: Initialized type system TS and time budget $T_{MAX}$
Result: $Q_{effective}$ and $Q_{crash}$: queues of effective and crashing test cases, respectively
1  $Q_{effective} \leftarrow \emptyset; Q_{crash} \leftarrow \emptyset;$Cov $\leftarrow \emptyset;$STraces $\leftarrow \emptyset$;
2  $DeterministicStage \leftarrow$ true;
3  $T_{start} \leftarrow$ Now();
4  while ElapsedTime($T_{start}$) $< T_{MAX}$ do
5      $tc \leftarrow$ LoadOrGenerateTestCase(TS, $Q_{effective}$, $DeterministicStage$);
6      if not $DeterministicStage$ then    /* Do not mutate TC during deterministic stage */
7          $tc \leftarrow$ MutateTC($tc$);
8      end
9      exe, $err \leftarrow$ BuildTempExe($tc$);
10     if $err = \emptyset$ then                                /* Build successful */
11         $ret_{code} \leftarrow$ Execute(exe);
12         if $ret_{code} = 0$ then                             /* Exited normally */
13             cov$_{tc} \leftarrow$ MeasureCoverage($tc$);
14             cov$_{new} \leftarrow$ cov$_{tc}$ − Cov;
15             if cov$_{new} \neq \emptyset$ then
16                 Cov $\leftarrow$ Cov $\cup$ cov$_{tc}$;
17                 $Q_{effective} \leftarrow Q_{effective} \cup \{tc\}$;
18             end
19         else                                                 /* Crash detected */
20             $out_{gdb} \leftarrow$ ExecuteInGDB($tc$);
21             $st_{trace} \leftarrow$ ParseStackTrace($out_{gdb}$);
22             if $st_{trace}$ not in STraces then
23                 STraces $\leftarrow$ STraces $\cup \{st_{trace}\}$;
24                 $Q_{crash} \leftarrow Q_{crash} \cup \{tc\}$;
25             end
26         end
27     end
28 end
```

the list of functions (L5 and L6). In the random stage, CLEMENTINE has two options to create a new test case: (1) generate a test case from scratch for a random target function, or (2) mutate previously generated test case from $Q_{effective}$. The probability of performing the first and second options is 50%.

Test Case Generation from Scratch. Algorithm 4 [5] shows how CLEMENTINE generates a test case from scratch for a target function. First, CLEMENTINE constructs an empty list of statements for the test case. Then, CLEMENTINE creates the test case by resolving all arguments in the target function. Function ResolveType at line 3 returns a variable op_{arg} with the requested type $type_{arg}$ given as the argument. CLEMENTINE has two following options to resolve a type:

[5]This algorithm is taken from [31] since CLEMENTINE uses the exact same algorithm as CITRUS.

Algorithm 3: `LoadOrGenerateTestCase`*

* This algorithm is directly taken from CITRUS [1, 31]

```
Data: Type system TS, queue of effective TCs $Q_{effective}$, and information of the current stage
      DeterministicStage
Result: A candidate test case $tc$ to be executed
1  if $DeterministicStage$ then                                        /* Deterministic stage */
2      $funcs \leftarrow$ AllFunctions(TS);
3      $f_{target} \leftarrow$ sequentially get a function from list of functions $funcs$;
4      $tc \leftarrow$ GenTCForMethod($f_{target}$);
5      if $f_{target} =$ EndOfList($funcs$) then                  /* end of deterministic stage */
6          $DeterministicStage \leftarrow$ false;
7      end
8  else                                                                /* Random stage */
9      $b_{gen\_new} \leftarrow$ RandInt($0, 1$);                           /* 50% prob */
10     if $Q_{effective}$ is empty or $b_{gen\_new} == 0$ then
11         $funcs \leftarrow$ AllFunctions(TS);
12         $f_{target} \leftarrow$ random function selected from $funcs$;
13         $tc \leftarrow$ GenTCForMethod($f_{target}$);
14     else
15         $tc \leftarrow$ RoundRobinSelection($Q_{effective}$);
16     end
17 end
18 return $tc$
```

Algorithm 4: `GenTCForMethod`*

* This algorithm is directly taken from CITRUS [1, 31]

```
Data: A target function $f$
Result: A test case $tc$ that calls $f$
1  $stmts \leftarrow \langle\rangle; args \leftarrow \langle\rangle$;
2  foreach $type_{arg}$ in ArgTypes($f$) do
3      $op_{arg} \leftarrow$ ResolveType($type_{arg}, stmts$);
4      $args \leftarrow args \cdot \langle op_{arg} \rangle$;
5  end
6  if $f$ needs invoking object then
7      $cls_f \leftarrow$ ClassOwner($f$);
8      $op_{inv} \leftarrow$ ResolveType($cls_f, stmts$);
9      $s_{call} \leftarrow$ CallWithInvokingObj($f, op_{inv}, args$);
10 else
11     $s_{call} \leftarrow$ Call($f, args$);
12 end
13 $stmts \leftarrow stmts \cdot \langle s_{call} \rangle$;
14 return MakeTC($stmts$)
```

1. Constructing a new variable. To construct a new variable, CLEMENTINE will create a new statement s which can be primitive variable type declaration (e.g., `int int1 = 39;`) or method invocation of object creator (e.g., `ClassA classa1 = ClassA::CreateObj();`). CLEMENTINE resolves the argument of the invoked method by recursively calling `ResolveType`. The new statement will be appended to $stmts$.

2. Reusing an existing variable. CLEMENTINE selects a statement that has a matching type with the required type from the existing statement in $stmts$.

If the target function f is a non-static member function of a specific class, CLEMENTINE will construct an instance of that specific type to call the target function. Lastly, CLEMENTINE appended the method call statement of target function f to $stmts$.

Algorithm 5: `MutateTC`*

* This algorithm is directly taken from CITRUS [1, 31]

```
Data: A CLEMENTINE test case $tc$ to mutate, $MAX$: a maximum number of mutations to $tc$
Result: The mutated test case $tc'$
1  $tc' \leftarrow tc$;
2  $n \leftarrow$ RandInt$(0, MAX)$;
3  for $i \leftarrow 1$ to $n$ do
4      switch RandInt$(0, 2)$ do
5          case 0 do
6              $tc' \leftarrow$ Randomly insert a random method call at a random position in $tc'$
7          case 1 do
8              $tc' \leftarrow$ Randomly mutate a statement in $tc'$
9          case 2 do
10             $tc' \leftarrow$ Delete unused variables in $tc'$
11     end
12 end
13 return $tc'$
```

Test Case Mutation. Algorithm 5 [6] (`MutateTC`) shows how CLEMENTINE creates a new test case by mutating an existing test case. There are 3 test case mutations in CLEMENTINE: (1) insert a random function call at a random position in tc', (2) mutate a random statement in tc', and (3) delete unused variables in tc'. CLEMENTINE randomly select one of those options to create a new test case. CLEMENTINE uses six mutation operators for mutating a random statement in tc'.

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] This algorithm is taken from CITRUS [1, 31] since CLEMENTINE uses the same algorithm.

6. OANG (Arithmetic Operator Negation).

Crash Deduplication. As shown in the Algorithm 2 at lines 21-25, CLEMENTINE does not save duplicate crashes. CLEMENTINE determines whether a crashing test case is unique from other crashing test cases by using the stack trace information [34]. CLEMENTINE compares the sequence of source code locations (i.e., file names + line number) obtained from the function call stack of the `gdb` stack trace. CLEMENTINE ignores the source code locations located outside the target program to avoid duplicate crashes caused by external libraries.

(3) Post-processing the generated test cases

The last step of CLEMENTINE is writing the effective test cases $Q_{effective}$ and crashing test cases Q_{crash}. CLEMENTINE writes the effective test cases $Q_{effective}$ in two different formats as follows:

1. `libfuzzer` test drivers.
2. Google Test test cases format.

CLEMENTINE also writes the unique crashing test cases in Q_{crash} including the stack trace obtained from `gdb` to help the user analyze the crash without re-executing the test case.

2.2.4 Implementation Details

CLEMENTINE is implemented using C++ programming language. It uses LLVM's LibTooling framework to parse C++ source code files and traverse the abstract syntax tree (AST). The current version of CLEMENTINE uses LLVM 11.0.1 and has been tested on Ubuntu 18.04 LTS versions. CLEMENTINE uses a modified version of LCOV [35] to get the coverage information of the target program.

CLEMENTINE has a command-line interface and it requires five command-line options. Those five required command-line options are as follows:

1. A preprocessed C++ source code file m_{ii}.
2. A path to a target program directory.
3. A path to the output directory.
4. A file containing all compile and link commands used to build the target program.
5. A path to an executable/library generated in the target program.

CLEMENTINE assumes the target program directory contains the compilation database of the target program (i.e., `compile_command.json`). The compilation database in the target program directory is used to get the information of necessary compilation flags to compile m_{ii}. CLEMENTINE will store the generated test cases in the output directory.

Types and Statements

As shown in Table 2.1, there are seven types in CLEMENTINE internal systems. The type may have zero or `type modifier` such as const or pointer (i.e., `*`). For `FunctionType`, CLEMENTINE will extract the function return's type and all its argument's type and make it as its *type*. For example, "`int addition(int a, int b);`" is a `int ()(int, int)` type.

Table 2.2 shows the five statement types in CLEMENTINE implementations. As mentioned in Subsection2.2.2, each statement has a type and might have a variable name.

Table 2.1: Type Categories*

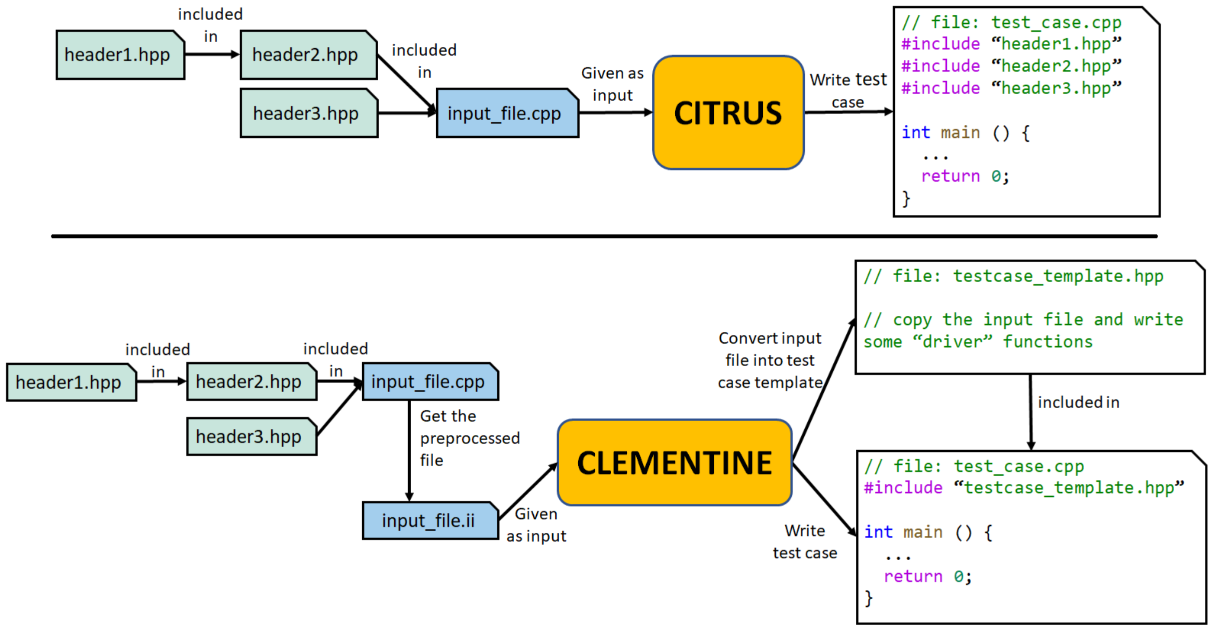
* This table is directly taken from CITRUS [1, 31]

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>`
`FunctionType`	Function type	`void (*)()`,
		`int (*)(int, int)`

Table 2.2: Statement Variants*

* This table is directly taken from CITRUS [1, 31]

Statement Name	**Representation**	**Examples**
`PrimitiveStmt`	Simple assignment	`int int1 = 39;`
`ArrayInitStmt`	Array initialization	`char[2] char2 {‘i’,‘q’};`
`CallStmt`	Function/constructor call	`ClassC classc3{char2};`
		`int int4 = class3.Invoke(int1);`
`STLStmt`	`STL` object construction	`std::vector<int,int> vec5 {{1, 2}};`
		`std::array<int,2> {0, 0};`
`DynAllocStmt`	Dynamic memory allocation	`void * void1 = malloc(32);`



```
// file: test_case.cpp
#include “header1.hpp”
#include “header2.hpp”
#include “header3.hpp”

int main () {
  ...
  return 0;
}
```

```
// file: testcase_template.hpp

// copy the input file and write
some “driver” functions
```

```
// file: test_case.cpp
#include “testcase_template.hpp”

int main () {
  ...
  return 0;
}
```

Figure 2.7: Difference between CITRUS and CLEMENTINE on writing test case

Writing Test Cases

As explained in subsection 2.1.1, including all header files is not a good way to write the test case. CLEMENTINE solved this problem by changing the input type from a cpp file into a preprocessed file and converting the preprocessed input file into a test case template. Figure 2.7 shows the difference between CITRUS and CLEMENTINE on writing the test case. Since the input type is changed (i.e., from a *.cpp* file to a preprocessed file *.ii*), the user needs to preprocess the cpp file first to get the preprocessed file using C++ preprocessor. Then, CLEMENTINE converts the preprocessed input file into a test case template by adding “driver” functions in the preprocessed input file. The test case template also contains functions that are not declared in the header file and including the test case template will allow the test case to call those functions.

Compile and Link

As explained in the Subsection 2.1.2, CITRUS requires the user to manually set up the linking configuration to test the target program and misconfiguration of this linking command will lead to ineffective testing performance. To avoid such linking configuration, CLEMENTINE utilizes the compile and link command used to build the target program. In exchange for linking configuration, CLEMENTINE requires 2 new inputs. The first new input is a text file that contains all commands (i.e., compile command and linking command) used to build the target program and the second new input is a path to an executable/library from the target program. The user can obtain the file that contains all commands from C++ build tools. For example, if the target subject’s build script is using Makefile, we can perform “dry run” (i.e., `make -n >> buildcmdfile`) to dump all commands used to build the target subject to a file named `buildcmdfile`. Figure 2.6 shows an example of such file. give an example of the build command file CLEMENTINE reads all commands in the file and recognizes the compile command and linking command. Then, CLEMENTINE will select the linking command that is used to generate the executable specified by the user in the second new input. This approach enables CLEMENTINE to ob-

```
 1:  template<typename T>
 2:  void templateFunc1(T x) { ... };
 3:
 4:  template<typename T>
 5:  void templateFunc2(T x) { ... };
 6:
 7: int use_templateFunc1 (bool cond) {
 8:     if (cond) {
 9:         bool x = false;
10:         templateFunc1<bool>(x);
11:     } else {
12:         int x = 10;
13:         templateFunc1<int>(x);
14:     }
15: }
```

Figure 2.8: Challenge in Instantiating Template Classes in C++

tain the linking configuration automatically to reduce user effort and the possibility of misconfiguration of the linking configuration

Testing C++ Template Classes/Functions

Testing templated classes and functions in C++ is difficult because it can be specialized with any type and it is not possible to test with all the possible types. However, some templated classes and functions also need to be specialized with a type that satisfies some specific characteristic. To solve this, CLEMENTINE binds a free template type to a concrete type based on the template specialization in the target program (if any). For example, consider the example in Figure 2.8. There are two templated functions in Figure 2.8, function `templateFunc1` at line 2 and function `templateFunc2` at line 5. In this example, there are two specializations for function `templateFunc1`. Function call at line 10 specialized function `templateFunc1` with `bool` type and function call at line 13 specialized function `templateFunc1` with `int` type. Therefore, CLEMENTINE randomly binds free template `T` with either `int` type or `bool` type when generating a test case for function `templateFunc1`. On the other hand, there is no specialization for function `templateFunc2`. In such case, CLEMENTINE randomly binds the free template `T` with `int` or `double` type.

Handling C++ STL Classes

STL, which stands for standard template library, is a collection of various data structures implemented using templated classes in C++. It is important to support STL classes because most of the C++ uses STL classes. However, handling STL classes is challenging since some STL classes have different ways to construct than others. Some STL classes have more complicated building methods than just the standard constructor-calling approach. For example, `std::tuple` should be constructed using function `std::make_tuple` rather than directly calling its own constructor.

To overcome such issues, CLEMENTINE has special treatment for each STL class. CLEMENTINE will not randomly select "object creator" of the STL class to construct an instance of STL class, instead CLEMENTINE has different treatment for each STL class. For example, CLEMENTINE uses

```
// file: header.hpp
class A{
 public:
  A() {};
 private:
  ~A() {}; // non-public destructor
};
```

Figure 2.9: Example of non-instantiable class

`std::make_pair` to construct an instance of `std::pair`. Meanwhile for `std::ostream`, CLEMENTINE simply gives `std::cout`.
There are 26 supported STL classes in this current version of CLEMENTINE. Those 26 supported STL classes can be divided into the following five categories:

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

2.3 Handling Complex C++ Features

I have analyzed functions that are not properly handled by CITRUS and classified them into 20 classes. I also grouped the 20 classifications into 4 groups based on their similarity. The first group is the "member function" group where the not-properly-handled function is a member function. The second group is the "scope/accessibility issue" group where functions are not properly handled due to scope or accessibility issues. The third group is the "unsupported type" group where functions are not properly handled due to unsupported type in CITRUS. The last group is "other" where the function did not meet the specific criteria of the previous groups Table 2.3 shows the 20 classifications of not-properly-handled functions and each status. "Solved" means CLEMENTINE is able to generate a test case to test such function. "Not necessary" means CLEMENTINE does not need to test such function. "Partial" means there are still some cases where CLEMENTINE can not handle. "Not yet" means CLEMENTINE still cannot handle such functions.

2.3.1 Constructor of Non-instantiable Class

In C++, there are some classes that cannot be instantiated directly. Figure 2.9 shows an example of non-instantiable classes. `Class A` is a non-instantiable class because it has a private destructor. Constructing an instance of `Class A` by calling its constructor like in Figure 2.10 at line 3 will cause a compilation error. Figure 2.11 shows the error message of compiling `main.cpp` in Figure 2.10. Thus, CLEMENTINE does not target the constructor of non-instantiable classes.

Figure 2.10 also shows another way of constructing an instance of `Class A` at line 4, that is creating a pointer to `Class A` using `new` operator. This way of constructing an instance of `Class A` does not cause

Table 2.3: Classification of Not-Properly-Handled Functions in CITRUS

Class ID	**Class Group**	**Class Name**	**Status**
1	Member Function	Constructor of non-instantiable class	Not necessary
2		Copy constructor of a class	Solved
3		Desctructor of a class	Solved
4		Implicit member function	Not necessary
5		Move constructor of a class	Solved
6		Pure virtual member function	Not necessary
7	Scope/Accessibility Issue	Function inside anonymous namespace	Solved
8		Function that is not declared in the header file	Solved
9		Function that requires enum type that declared inside anonymous namespace	Solved
10		Non-public member function	Solved
11		Static function	Solved
12	Unsupported Type	Function related to unrecognized class type	Not yet
13		Function that has void pointer argument type	Solved
14		Function that has function pointer argument type	Solved
15		Function that has unhandled clang type	Not yet
16		Unhandled STL	Partial
17	Other	Functions related to a class whose “object constructors” were not detected	Not yet
18		Global operator overloading	Solved
19		Inline function without definition	Not necessary
20		Template function without definition	Not necessary

```
// file: main.cpp
1: #include "header.hpp"
2: int main() {
3:  A a; // compile error
4:  // A *ptr_a = new A();  // successfully compiled
5: }
```

Figure 2.10: Example of constructing an instance non-instantiable class

```
main.cpp:4:4: error: variable of type 'A' has private destructor
 A a;
   ^
./header.hpp:6:5: note: declared private here
    ~A();
    ^
1 error generated.
```

Figure 2.11: Error message of constructing a non-instantiable class

```
// file: testcase_template.hpp
1: class A {
2:  public:
3:   A(int x) { value = x; };
4:   A(const A &x) { value = x.value; }; // copy constructor
5:  private:
6:   int value;
7: };
```

Figure 2.12: Example of copy constructor function

```
// file: test_case.cpp
1: #include "testcase_template.hpp"
2: int main () {
3:  int int1 = 10;
4:  A a2(int1);
5:  A a3(a2); // copy constructor is called here
6:  return 0;
7: }
```

Figure 2.13: Example of test case that tests copy constructor function

a compilation error. This can be an alternative way to construct an instance of non-instantiable class in the next version of CLEMENTINE.

2.3.2 Copy Constructor Function

Copy constructor is a member function that is used to create a new class instance and initialize its

```cpp
// file: testcase_template.hpp
class A {
 public:
  A(int x) { value = x; };
  ~A() { }; // destructor of class A
  int getValue() { return value; };
 private:
  int value;
};
```

Figure 2.14: Example of destructor function

```cpp
// file: test_case.cpp
#include "testcase_template.hpp"
int main () {
 int int1 = 10;
 A a2(int1); // instantiate object of class A
 return 0; // return statement to immediately exit the main function
 // destructor of class A will be called when a2 goes out of scope
}
```

Figure 2.15: Example of a test case to test destructor function

value with an existing object of the same class. The compiler will always generate copy constructor for a class, struct, and union if there is no user-defined copy constructor [36]. Figure 2.12 shows an example of a user-defined copy constructor. Previously, CITRUS did not target copy constructor of a class.

To test copy constructor, CLEMENTINE generates a test case that creates an instance of a class by calling the constructor and creates a copy of the instance by calling copy constructor. CLEMENTINE does not consider copy constructor as an "object creator"[7] because it requires an object with the same class and it may cause infinite recursion. Figure 2.13 shows the example of a test case that tests copy constructor.

2.3.3 Destructor Function

In C++, the destructor of a class is implicitly called when an instance of a class goes out of scope. Although it is implicitly called, there is no guarantee that the destructor function will be called. For example, the destructor of a class is not called if the test case is crash before an instance of a class goes out of scope. Thus, it is better to test the destructor of a class individually. However, CITRUS did not target such function. Figure 2.14 shows the example of a destructor function.

To test the destructor function, CLEMENTINE generates a test case that creates an instance of a class and immediately exits the main function. Immediately exiting the main function after instantiating an object of a class is to make sure the destructor of the class is called unless there is an error or crash in the constructor. Figure 2.15 shows an example of a test case that tests the destructor of class A.

[7]Function that is used to provide an instance of a class. See Subsection 2.2 for the detail.

```cpp
// file: testcase_template.hpp
class A {
 public:
  int x;
  // The compiler implicitly generates several special member functions for class A
};
```

Figure 2.16: Example of implicit function

```cpp
// file: main.cpp
1: #include "testcase_template.cpp"
2: int main() {
3:   A a; // successfully compiled
4:   return 0;
5: };
```

Figure 2.17: Example of calling implicit function

```cpp
// file: testcase_template.hpp
class A {
 public:
  A(int x) { value = new int; *value = x; };
  A(A &&x) : A{ *x.value } { x.value = nullptr; }; // move constructor
 private:
  int * value;
};
```

Figure 2.18: Example of move constructor function

2.3.4 Implicit Member Function

C++ compiler implicitly generates a definition for several special member functions if they are not explicitly defined in the source code. Those special member functions are default constructor, destructor, copy constructor, move constructor, and assignment operator (i.e., copy assignment and move assignment) [37]. Figure 2.16 shows an example of a class in which the compiler will implicitly generate several special member functions for it. Figure 2.17 shows an example of a test case that calls a default constructor of `Class A` which is an implicit function at line 3. Although the `main.cpp` is compilable, testing the implicit function does not increase the test coverage. Moreover, the implicit function does not have actual source code written by the developer. Thus, CLEMENTINE intentionally does not target implicit function since the implicit function.

2.3.5 Move Constructor Function

Move constructor is a member function that is used to create a new class instance by transferring the ownership of resources from an existing object with the same class. By transferring the ownership of resources, move constructor avoids unnecessarily copying data in the memory. Figure 2.18 shows an

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main () {
 int int1 = 10;
 A a2(int1);
 A a3(std::move(a2)); // move constructor is called here
 return 0;
}
```

Figure 2.19: Example of test case that tests move constructor function

```
// file: testcase_template.hpp
1: class Parent {
2:  public:
3:   virtual void fun() = 0; // pure virtual member function
4:   int getX() {return x};
5:  private:
6:   int x;
7: };
```

Figure 2.20: Example of pure virtual function

```
// file: main.cpp
#include "testcase_template.hpp"
int main() {
 Parent x; // not allowed to construct abstract class (i.e., Parent)
 x.fun(); // not allowed to call pure virtual function (i.e., fun)
 return 0;
}
```

Figure 2.21: Example of calling pure virtual function

example of a user-defined move constructor. Previously, CITRUS did not target the move constructor of a class.

To test the move constructor, CLEMENTINE generates a test case that creates an instance of a class and then calls the move constructor. CLEMENTINE uses the standard C++ library function (i.e., `std::move`) to convert an object to an rvalue reference to that object. Figure 2.19 shows the example of a test case that tests the move constructor. Similar to copy constructor, CLEMENTINE also does not consider move constructor as an "object creator" because it requires an object with the same class and it may cause infinite recursion.

2.3.6 Pure Virtual Member Function

In C++, a class that has pure virtual function is called an abstract class and it cannot be instantiated [38]. A member function is a pure virtual function if it is declared with "`virtual`" and followed by "`=0`" Figure 2.20 shows the example of a pure virtual function at line 3(i.e., `fun`). A pure virtual function cannot have a definition and it has to be overridden in the derived class. Constructing an abstract

```
main.cpp:3:10: error: variable type 'Parent' is an abstract class
  Parent x;
         ^
./header.hpp:3:18: note: unimplemented pure virtual method 'fun' in 'Parent'
    virtual void fun() = 0;
                 ^
1 error generated.
```

Figure 2.22: Error message of calling pure virtual function

```
// file: target_file.hpp (header file)
namespace { // anonymous namespace or unnamed namespace
 void anonymousNamespaceFunction(); // only declaration
}
```

```
// file: target_file.cpp (implementation/cpp file)
#include "target_file.hpp"
namespace { // anonymous namespace or unnamed namespace
 void anonymousNamespaceFunction() { // function inside anonymous namespace
 }
}
```

Figure 2.23: Example of function inside anonymous namespace

```
// file: main1.cpp
#include "target_file.hpp" // include the header file
int main () {
 anonymousNamespaceFunction(); // not allowed, since function
 anonymousNamespaceFunction scope is limited to target_file.cpp
 return 0;
}
```

```
// command to build main1.cpp
clang++ -c target_file.cpp -o target_file.o -O0 -g --save-temps --coverage
clang++ -c main1.cpp -o main1.o -O0 -g --save-temps --coverage
clang++ -o exemain main1.o target_file.o -g -O0 --save-temps --coverage
```

Figure 2.24: Incorrect way of testing function inside anonymous namespace

class and calling its pure virtual function like in Figure 2.21 at line 3 and line 4 respectively, will cause a compilation error. Figure 2.22 shows the error message obtained while trying to compile `main.cpp`. Thus, CLEMENTINE does not target pure virtual function because calling pure virtual function will hinder the testing performance.

2.3.7 Function Inside Anonymous Namespace

Namespace is a C++ feature that can be used to avoid naming conflicts in a large project. A namespace that is defined without an identifier is called an anonymous namespace (a.k.a., unnamed

```
main1.o: In function 'main':
main1.cpp:3: undefined reference to '(anonymous
    namespace)::anonymousNamespaceFunction()'
clang-11: error: linker command failed with exit code 1 (use -v to see invocation)
```

Figure 2.25: Error message of calling a function inside anonymous namespace

```
// file: main2.cpp
1: #include "target_file.cpp" // include the implementation/header file
2: int main () {
3:   anonymousNamespaceFunction(); // allowed, since main1.cpp include
    target_file.cpp, thus main1.cpp also has definition of
    anonymousNamespaceFunction
4:   return 0;
5: }
```

```
// command to build main2.cpp
clang++ -c main2.cpp -o main2.o -O0 -g --save-temps --coverage
clang++ -o exemain main2.o -g -O0 --save-temps --coverage
```

Figure 2.26: Correct way of testing function inside anonymous namespace

namespace [39]). The anonymous namespace is used to make sure everything defined in the namespace has internal linkage (i.e., the scope is limited to one translation unit). Figure 2.23 shows an example of a function inside anonymous namespace at line 3(i.e., `anonymousNamespaceFunction`). The scope of `anonymousNamespaceFunction` is limited to `target_file.cpp` and cannot be called from other translation unit files. Therefore, `anonymousNamespaceFunction` cannot be called from `main1.cpp` like in the Figure 2.24 at line 3. Figure 2.25 shows the error message obtained during linking the `main1.cpp`.

CLEMENTINE solves this by changing the way of writing test cases (i.e., create a test case template from the preprocessed file and include it in the test case file). Figure 2.26 shows the correct way of testing functions inside anonymous namespace. `anonymousNamespaceFunction` can be called from `main2.cpp` because `main2.cpp` includes the implementation file (i.e., `target_file.cpp`) not the header file. Thus, `main2.cpp` also has the definition of `anonymousNamespaceFunction`. Note that the linking command need to be changed since `main2.cpp` cannot be linked with `target_file.cpp`. Linking `main2.cpp` with `target_file.cpp` will cause "`multiple definition problem`". Figure 2.26 also shows the linking command to build `main2.cpp`.

2.3.8 Function That is not Declared in The Header File

In C++, declarations of a class or a function are not always located in the header file. The developer of C++ programs may write class declarations in the implementation file (i.e., cpp file). Classes and functions whose declarations are written in the implementation file are intended to be used locally not globally (i.e., cannot be used from other implementation files). Function `funcDeclaredInImplFile` at line B2 in Figure 2.27 is the example of such function. Therefore, calling function `funcDeclaredInImplFile` at line 3 in the Figure 2.28 is not possible since `main1.cpp` does not have declaration of `funcDeclaredInImplFile` (i.e., `main1.cpp` includes only the header file `target_file.hpp` and it does not contain declaration of `funcDeclaredInImplFile`). Figure 2.29 shows the error message during the compilation of `main1.cpp`.

```
// file: target_file.hpp (header file)
A1: // no declaration of funcDeclaredInImplFile
```

```
// file: target_file.cpp (implementation/cpp file)
B1: #include "target_file.hpp"
B2: void funcDeclaredInImplFile() { }
```

Figure 2.27: Example of function that is not declared in the header file

```
// file: main1.cpp
1: #include "target_file.hpp" // include the header file
2: int main () {
3:  funcDeclaredInImplFile(); // not allowed, since main1.cpp does not have
    declaration of funcDeclaredInImplFile
4:  return 0;
5: }
```

```
// command to build main1.cpp
clang++ -c target_file.cpp -o target_file.o -O0 -g --save-temps --coverage
clang++ -c main1.cpp -o main1.o -O0 -g --save-temps --coverage
clang++ -o exemain main1.o target_file.o -g -O0 --save-temps --coverage
```

Figure 2.28: Incorrect way of testing function that is not declared in the header file

```
main1.cpp:3:3: error: use of undeclared identifier 'funcDeclaredInImplFile'
  funcDeclaredInImplFile();
  ^
1 error generated.
```

Figure 2.29: Error message of calling function that is not declared in the header file

```
// file: main2.cpp
1: #include "target_file.cpp" // include the implementation/header file
2: int main () {
3:  funcDeclaredInImplFile(); // allowed, since main1.cpp include target_file.cpp,
    thus main1.cpp also has declaration of funcDeclaredInImplFile
4:  return 0;
5: }
```

```
// command to build main2.cpp
clang++ -c main2.cpp -o main2.o -O0 -g --save-temps --coverage
clang++ -o exemain main2.o -g -O0 --save-temps --coverage
```

Figure 2.30: Correct way of testing function that is not declared in the header file

Therefore, the way of CITRUS writes the test case (i.e., includes all header files in the test case) cannot test such functions.

CLEMENTINE solves this by changing the way of writing test cases (i.e., create a test case template from the preprocessed file and include it in the test case file). Calling function `funcDeclaredInImplFile`

```
// file: target_file.hpp (header file)
A1: namespace { // anonymous namespace
A2:   enum class A { // enum type that declared in anonymous namepsace
A3:     kOne = 0,
A4:     kTwo,
A5:   };
A6: }
A7: void useEnumA(A x); // function that need enum type that declared inside
    anonymous namespace
```

```
// file: target_file.cpp (implementation/cpp file)
B1: #include "target_file.hpp"
B2: #include <iostream>
B3: void useEnumA(A x) {
B4:   std::cout << "useEnumA" << "\n";
B5: }
```

Figure 2.31: Example of function that requires `enum` type that declared inside anonymous namespace

```
// file: main1.cpp
1: #include "target_file.hpp" // include the header file
2: int main() {
3:   useEnumA(A::kOne); // not allowed, since it requires enum type "A" that
    declared inside anonymous namespace in target_file.cpp
4:   return 0;
5: }
```

```
// command to build main1.cpp
clang++ -c target_file.cpp -o target_file.o -O0 -g --save-temps --coverage
clang++ -c main1.cpp -o main1.o -O0 -g --save-temps --coverage
clang++ -o exemain main1.o target_file.o -g -O0 --save-temps --coverage
```

Figure 2.32: Incorrect way of testing function that requires `enum` type that declared inside anonymous namespace

at line 3 in the Figure 2.30 is fine because `main2.cpp` includes `target_file.cpp` which contains definition of `funcDeclaredInImplFile`. Note that the linking command need to be changed since `main2.cpp` cannot be linked with `target_file.cpp`. Figure 2.30 also shows the linking command to build `main2.cpp`.

2.3.9 Function That Requires `enum` Type That Declared Inside Anonymous Namespace

As explained in subsection 2.3.7, everything defined inside anonymous namespace has internal linkage. Thus, the scope of the enum defined in the anonymous namespace is limited to one translation unit file Figure 2.31 shows the example of an `enum` that is declared inside anonymous namespace at line A2 (i.e., enum `A`) and a function that uses the `enum` type at line A7 (i.e., `useEnumA`). Function `useEnumA` cannot be called at line 3 in Figure 2.32 because `main1.cpp` does not have definition of function `useEnumA`. Figure 2.33 shows the error message of compiling `main1.cpp`.

```
In file included from main1.cpp:1:
./target_file.hpp:7:6: error: function 'useEnumA' is used but not defined in this
    translation unit, and cannot be defined in any other translation unit because
    its type does not have linkage
void useEnumA(A x);
     ^
main1.cpp:3:3: note: used here
  useEnumA(A::kOne);
  ^
1 error generated.
```

Figure 2.33: Error message of calling function that requires `enum` type that declared in anonymous namespace

```
// file: main2.cpp
1: #include "target_file.cpp" // include the implementation file
2: int main() {
3:   useEnumA(A::kOne); // not allowed, since it requires enum type "A" that
    declared inside anonymous namespace in target_file.cpp
4:   return 0;
5: }
```

```
// command to build main2.cpp
clang++ -c main2.cpp -o main2.o -O0 -g --save-temps --coverage
clang++ -o exemain main2.o -g -O0 --save-temps --coverage
```

Figure 2.34: Correct way of testing function that requires `enum` type that declared inside anonymous namespace

CLEMENTINE solves this by including the preprocessed input file in the test case to make sure the test case has the definition of the `enum` type and the function that requires the `enum` type. Calling function `useEnumA` at line 3 in the Figure 2.34 is fine because `main2.cpp` has the definition of `useEnumeA` and `enum` type `A` since it includes `target_file.cpp`. Note that the linking command need to be changed since `main2.cpp` cannot be linked with `target_file.cpp`. Figure 2.34 also shows the linking command to build `main2.cpp`.

2.3.10 Non-public Member Function of a Class

C++ has access specifiers to control the visibility and accessibility of a class, struct, or union member (both variable and function). There are 3 access specifiers [40], which are: (1) public, (2) protected, and (3) private.Public members are accessible from anywhere in the program even outside of the class. Private members are accessible only within the class and cannot be accessed directly from outside the class and the derived class. Protected members are accessible within the class and the derived class. Figure 2.35 shows the example of non-public member functions.

Testing public member functions is not difficult since they can be accessed publicly (i.e., accessible from anywhere in the program). However, testing non-public (i.e., private and protected) member functions is not as simple as testing public member functions since non-public member functions can only

```
// file: target_file.hpp (header file)
1: class A {
2:   public:
3:     void publicFunc();
4:   private:
5:     void privateFunc(); // private function
6: };
```

```
// file: target_file.cpp (implementation/cpp file)
1: #include "target_file.hpp"
2: void A::publicFunc() {}
3: void A::privateFunc() {}
```

Figure 2.35: Example of non-public member function

```
// file: main1.cpp
1: #include "target_file.hpp"
2: int main () {
3:  A a1;
4:  a1.privateFunc();
5:  return 0;
6: }
```

```
// command to build main1.cpp
clang++ -c target_file.cpp -o target_file.o -O0 -g --save-temps --coverage
clang++ -c main1.cpp -o main1.o -O0 -g --save-temps --coverage
clang++ -o exemain main1.o target_file.o -g -O0 --save-temps --coverage
```

Figure 2.36: Incorrect way of testing non-public member function

```
main1.cpp:4:5: error: 'privateFunc' is a private member of 'A'
  a.privateFunc();
    ^
./target_file.hpp:5:10: note: declared private here
    void privateFunc();
         ^
1 error generated.
```

Figure 2.37: Error message of calling non-public member function in the test case file

be accessed within the class. Calling non-public functions directly in the test case like in Figure 2.36 at line 4 will cause a compilation error. Figure 2.37 shows the compile error message of compiling `main1.cpp`.

CLEMENTINE solves this accessibility problem by writing the "driver" function for each non-public member function in the test case template [8]. A "driver" function is a public function that calls only a non-public member function. The return type and argument type of the "driver" function are exactly the same as the return type and argument type of the non-public member function. Figure 2.38 shows

[8]Note that CLEMENTINE converts preprocessed input file into test case template by copying the preprocessed input file and writing "driver" function. See subsection 2.1.1.

```
// file: testcase_template.hpp
class A {
 public:
  void publicFunc();
 private:
  void privateFunc();
 public:
  void __driver_privateFunc() { return privateFunc(); }; // "driver" function
};
```

Figure 2.38: Example of test case template that contains "driver" function

```
// file: main2.cpp
#include "testcase_template.hpp"
int main () {
 A a1;
 a1._bypass_privateFunc();
 return 0;
}
```

```
// command to build main2.cpp
clang++ -c main2.cpp -o main2.o -O0 -g --save-temps --coverage
clang++ -o exemain main2.o -g -O0 --save-temps --coverage
```

Figure 2.39: Example of test case that tests non-public member function

```
// file: target_file.hpp (header file)
static function funA();
```

```
// file: target_file.cpp (implementation/cpp file)
#include "target_file.hpp"
static function funA() {}
```

Figure 2.40: Example of static function

the example of "driver" functions written by CLEMENTINE. Since the "driver" function is a public function, it can be called directly in the test case file and it is used as a proxy to test non-public member function. Figure 2.39 shows the example of a test case that tests non-public member function generated by CLEMENTINE. Note that the linking command to build `main2.cpp` should be changed because `main2.cpp` cannot be linked with `target_file.cpp`. Figure 2.39 also shows the linking command to build `main2.cpp`.

2.3.11 Static Function

In C++, a function can be declared as a static function using a "static" keyword like the code in Figure 2.40. Previously, CITRUS did not target static function because the scope of static function is limited to one translation unit (i.e., cpp file) where the static function is located. Function `funA` cannot be called in the `main1.cpp` at line 3 in the Figure 2.41 because the scope of `funA` is limited to one

```
// file: main1.cpp
1: #include "target_file.hpp" // include the header file
2: int main() {
3:  funA(); // not allowed, because funA scope is limited to 1 translation unit
    (i.e., target_impl.cpp)
4:  return 0;
5: }
```

```
// command to build main1.cpp
clang++ -c target_file.cpp -o target_file.o -O0 -g --save-temps --coverage
clang++ -c main1.cpp -o main1.o -O0 -g --save-temps --coverage
clang++ -o exemain main1.o target_file.o -g -O0 --save-temps --coverage
```

Figure 2.41: Incorrect way of testing static function

```
main1.o: In function 'main':
main1.cpp:3: undefined reference to 'funA()'
clang-11: error: linker command failed with exit code 1 (use -v to see invocation)
```

Figure 2.42: Error message of calling static function in `main1.cpp`

```
// file: main2.cpp
1: #include "target_file.cpp" // include the implementation file
2: int main() {
3:  funA(); // allowed, because main2.cpp includes the impl.cpp, thus it contains
    the definition of funA
4:  return 0;
5: }
```

```
// command to build main2.cpp
clang++ -c main2.cpp -o main2.o -O0 -g --save-temps --coverage
clang++ -o exemain main2.o -g -O0 --save-temps --coverage
```

Figure 2.43: Incorrect way of testing static function

translation unit (i.e., `target_file.cpp`). Thus, build process of `main1.cpp` will fail due to "undefined reference" error as shown in the Figure 2.42.

CLEMENTINE solves it by changing the way it writes the test case (see subsection 2.1.1 for the explanation). Including the preprocessed input file in the test case will make the test case has the definition of static function. Thus, calling `funA` in `main2.cpp` at line 3 in the Figure 2.43 is fine. However, the command to build `main2.cpp` should be changed since `main2.cpp` cannot be linked with `target_file.cpp`. Figure 2.43 also shows the command to build `main2.cpp`.

2.3.12 Function Related to Unrecognized Type

Before starting the method sequence generation, CLEMENTINE creates the program representation of the target subject. In this stage, CLEMENTINE collects information of `classes`, `structs`, `enums`, and functions declared in the target program. CLEMENTINE also removes all unsatisfiable functions from

```
// file: target_file.cpp
 1: #include <cstdio>
 2: void writeToFile(FILE* f, std::string message) {}
 3: class OuterClass {
 4:  private:
 5:   class NestedClass { // private nested class
 6:    public:
 7:     int x;
 8:   };
 9:   void useNestedClass(NestedClass& nestedObj) {} // function that requires an instance of private nested class
10: };
```

Figure 2.44: Example of function related to unrecognized type

```
// file: testcase_template.hpp
void funcWithVoidPtr(void * x) { // function that requires void pointer type argument
 int *int_ptr = (int *) x;
 std::cout << "funcWithVoidPtr: " << *int_ptr << "\n";
}
```

Figure 2.45: Example of function that has void pointer type argument

the list of functions. A function is unsatisfiable if the function required an instance of a type which CLEMENTINE cannot construct. One of the reasons CLEMENTINE cannot construct an instance of a type is when CLEMENTINE does not recognize the type.

So far, there are 2 known reasons why CLEMENTINE does not recognize a type. The first reason is that the unrecognized type is a C++ standard library type which CLEMENTINE does not support. Function `writeToFile` at line 2 in Figure 2.44 is an example of such a function. The second reason is that the unrecognized type is a non-public nested class. Currently, CLEMENTINE does handle non-public nested classes [9]. Function `useNestedClass` is at line 10 an example of such function.

2.3.13 Function That Has Void Pointer Type Argument

In C++, void pointer (i.e., "`void *`") is a pointer that points to an object of an unknown type. Void pointer is useful to deal with memory addresses without information on the actual data type. Thus, it is hard to automatically generate a test case to test a function with void pointer argument type since it can be cast to any other pointer type. Lisitng 2.45 shows the example of a function that has void pointer type argument. CLEMENTINE tests such function by allocating a random-sized memory and passing it to the function as an argument. Lisitng 2.46 shows the example test case that tests function with void pointer type argument.

2.3.14 Function That Has Function Pointer Type Argument

[9]A future version of CLEMENTINE will support more standard C++ library types and C++ complex features.

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main() {
 void * void1 = malloc(20); // allocate random-sized memory
 funcWithVoidPtr(void1);
 free(void1);
 return 0;
}
```

Figure 2.46: Example of test case that test function with void pointer type argument

```
// file: testcase_template.hpp
1: int add(int a, int b){
2:   return a + b;
3: }
4: int addOne(int a){
5:   return a + 1;
6: }
7: void funcWithFuncPtr(int (*arg1)(int, int) ) { // function that requires
    function pointer as argument
8:   std::cout << arg1(5,2) << "\n";
9: }
```

Figure 2.47: Example of the function that has function pointer argument type

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main() {
 funcWithFuncPtr(add);
 return 0;
}
```

Figure 2.48: Example of test case that test function with function pointer argument type

Besides void pointer, C++ also has another special pointer type which is function pointer. A function pointer is a pointer that holds the address of a function (i.e., point to a function). Function pointer gives the ability to store and invoke functions dynamically at runtime. It is usually used in the callbacks mechanism. Figure 2.47 shows the example of a function with function pointer type argument. Previously, CITRUS did not target such a function because it did not support function pointer type.

CLEMENTINE solves it by supporting function pointer type. CLEMENTINE will search for a function in the target subject that has a matching function signature with the function pointer argument. For example in Figure 2.47, there is a function with function pointer argument type at line 7 (i.e., function `funcWithFuncPtr`). That function requires a pointer to a function that takes two integers and returns an integer. Then, CLEMENTINE will search for a function in the target subject that takes two integers and returns an integer, and give it as an argument to function `funcWithFuncPtr`. In this example, CLEMENTINE will give function `add` as an argument since it has a matching signature with the function

```
// file: target_file.hpp
 1: class BasicClass {
 2:  public:
 3:   BasicClass(int x) {value = x;};
 4:  private:
 5:   int value;
 6: };
 7: template <typename T>
 8: class TmplClass {
 9:  public:
10:   TmplClass(T x) {value = x;};
11:  private:
12:   T value;
13: };
14: void func1(BasicClass x) { // clang::CXXRecordDecl
15: };
16: void func2(TmplClass<int> x) { // clang::ClassTemplateSpecializationDecl
17: };
18: template <typename T>
19: void func3(T x) { // clang::TemplateTypeParmType
20: };
21: template <typename T>
22: void func4(TmplClass<T> x) { // clang::TemplateSpecializationType
23: };
24: void func5(int BasicClass::*pointerToMember) { // clang::MemberPointerType
25: };
```

Figure 2.49: Example of function that has unhandled clang class type

pointer and will not give function `addOne` since the function signature is different. Figure 2.48 shows the example of the generated test case that tests function with function pointer argument type.

2.3.15 Function That Has Unhandled Clang Class Type

CLEMENTINE uses clang [41] to parse the source code of the target subject and traverse the AST (i.e., abstract syntax tree). It also utilizes clang class type to recognize function parameter type. Figure 2.49 shows the example of how CLEMENTINE utilizes clang class type to recognize the type of function arguments. `clang::CXXRecordDecl` represents a class or struct type. A templated class that has been specialized is represented using `clang::ClassTemplateSpecializationDecl`. In this example, the templated class is `TmplClass` and it has been specialized with int. `clang::TemplateTypeParmType` represents template type.

There are 2 known unhandled clang class types by CLEMENTINE. Those 2 known unhandled clang class types are (1) `clang::TemplateSpecializationType` and (2) `clang::MemberPointerType`. Clang class `clang::TemplateSpecializationType` represents a templated class that has not been specialized as shown in the example in Figure 2.49 at line 25. `clang::MemberPointerType` represents a pointer that points to a variable member or member function. In the example shown in Figure 2.49, function `func5` requires a pointer to (variable) member of `BasicClass` that has type `int`. This type of function is

```
// file: testcase_template.hpp
#include <iostream>
#include <sstream>
#include <functional>
void printToOstream(std::ostream& outStream, std::string msg) {
  outStream << msg;
}
void printToOstringstream(std::ostringstream& stringStream, std::string msg) {
  stringStream << msg;
}
void needSTDFunction(std::function<int (int, int)> stlarg) {
  std::cout << "needSTDFunction: " << stlarg(2,3) << "\n";
}
```

Figure 2.50: Example of function that has unsupported STL type argument

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main() {
  std::ostream& x = std::cout;
  printToOstream(x, "message");
  return 0;
}
```

Figure 2.51: Example of test case that test function that need `std::ostream` argument

considered an unsatisfiable function and is removed from the target function by CLEMENTINE.

2.3.16 Unsupported STL

There are several STL types that are still not supported by CITRUS. So far, there are 6 known unsupported STL types [10]. Those 6 known unsupported STL types are: (1) `std::basic_ostream` / `std::ostream`, (2) `std::basic_istream` / `std::istream`, (3) `std::function`, (4) `std::initializer_list`, (5) `std::ostringstream`, and (6) `std::_List_iterator`. Figure 2.50 shows some examples of functions that require unsupported STL types. Among 6 known unsupported STL types, CLEMENTINE adds support to 3 STL types. Those 3 newly supported STL types are: (1) `std::basic_ostream` / `std::ostream`, (2) `std::basic_istream` / `std::istream`, and (3) `std::ostringstream`. CLEMENTINE pass `std::cout`, `std::cin`, and an instance of `std::ostringstream` to `std::basic_ostream`, `std::basic_istream`, and `std::ostringstream` respectively. Figure 2.51 shows an example of a test case that tests a function that has `std::ostream` argument.

2.3.17 Functions Related to a Class whose "Object Creator" functions were not detected

As mentioned in subsection 2.3.12, CLEMENTINE removes all unsatisfiable functions from the list of functions and an unsatisfiable function is a function that requires instance of type which CLEMENTINE

[10] These known unsupported STL types are obtained by applying CITRUS on 16 target subjects.

```
// file: tinyxml2.ii
 1: class TINYXML2_LIB XMLElement : public XMLNode {
 2:  friend class XMLDocument;
 3:  public:
 4:   const char* Name() const;
 5:   void SetName( const char* str, bool staticMem=false );
 6:   ...
 7:  private:
 8:   XMLElement(XMLDocument* doc); // non-public constructor
 9:   virtual ~XMLElement(); // non-public destructor
10:  ...
11: }
12: ...
13: class TINYXML2_LIB XMLDocument : public XMLNode {
14:  ...
15:  friend class XMLElement;
16:  public:
17:    XMLElement* NewElement(const char* name);
18:    XMLElement* RootElement();
19:  ...
20: }
```

Figure 2.52: Example of class whose "object creator" functions were not detected

cannot construct. Another reason CLEMENTINE cannot construct an instance of a type is when CLEMENTINE failed to recognize "object creator" of the type. Previously, CITRUS considers only public constructor and *static factory* method (i.e., public functions that are `static` and return a particular class type) as "object creator". At Figure 2.52, the constructor of `XMLElement` is not considered as "object creator" by CITRUS because it is not a public function. Function `NewElement` and function `RootElement` at line 17 and 18 are also not considered as "object creator" because those functions are not `static` functions. Therefore, CITRUS did not detect any "object creator" for class `XMLElement`. As a result, all member functions of `XMLElement` and all functions with `XMLElement` argument type are removed from the list of functions.

CLEMENTINE tries to solve this by redefining the definition of "object creator". The new definition of "object creator" is explained in Subsection 2.2. Based on the new definition, function `NewElement` and function `RootElement` are considered as "object creator" of `XMLElement` by CLEMENTINE. Thus, all member functions of `XMLElement` and all functions with `XMLElement` argument type are not removed from the list of functions.

2.3.18 Global Operator Overloading

Customizing operators for operands of user-defined types is called operator overloading. Operator overloading allows the developers to write intuitive syntax for operations on the user-defined type making the code more readable. Figure 2.53 shows an example of an operator overloading function. Operator overloading can be declared as member functions of a class and non-member functions depending on the operator. There are four operators that should be overloaded as member functions (i.e., `operator=`, `operator[]`, `operator()`, and `operator->`)[42]. While the other operators (e.g., `operator+`, `operator-`,

```
// file: testcase_template.hpp
class Number {
 private:
  int val;
 public:
  Number(int x) { val = x; };
  int getVal() { return val; };
};
Number operator+(Number& v1, Number& v2) { // global operator overloading
 int new_val = v1.getVal() + v2.getVal();
 return Number(new_val);
}
```

Figure 2.53: Example of global operator overloading function

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main () {
 Number number1(3);
 Number number2(9);
 Number number3 = number1 + number2;
 return 0;
}
```

Figure 2.54: Example of global operator overloading function

```
// file: testcase_template.hpp
inline void nodefInlineFunc();
```

Figure 2.55: Example of inline function without definition

```
// file: test_case.cpp
#include "testcase_template.hpp"
int main() {
  nodefInlineFunc(); // not allowed since there is no definition of "nodefInlineFunc"
  return 0;
}
```

Figure 2.56: Example of calling inline function without definition

`operator*`, etc.) can be declared globally (i.e., as non-member functions). Previously, CITRUS did not target global operator overloading functions. Now, CLEMENTINE targets the global operator overloading and Figure 2.54 shows the generated test case that test global operator overloading.

2.3.19 Inline Function without Definition

```
test_case.o: In function 'main':
test_case.cpp:3: undefined reference to 'nodefInlineFunc()'
clang-11: error: linker command failed with exit code 1 (use -v to see invocation)
```

Figure 2.57: Error message of calling inline function without definition

```
// file: testcase_template.hpp
1: Template <typename T>
2: void templateFuncWODefintion(T x);
```

Figure 2.58: Example of template function without definition

```
// file: test_case.cpp
1: #include "testcase_template.hpp"
2: int main() {
3:  templateFuncWODefintion<int>(10); // not allowed, since there is no definition
    of "templateFuncWODefintion"
4:  return 0;
5: }
```

Figure 2.59: Example of calling template function without definition

```
test_case.o: In function 'main':
test_case.cpp:3: undefined reference to 'void templateFuncWODefintion<int>(int)'
clang-11: error: linker command failed with exit code 1 (use -v to see invocation)
```

Figure 2.60: Error message of calling template function without definition

Inline function is a C++ feature used to reduce the execution time [43]. The "inline" keyword is used to declare a function as an inline function. Every function call to an inline function will be replaced with its function definition during compile or linking time. Figure 2.55 shows an example of an inline function without definition. Calling an inline function that does not have a definition like in the Figure 2.56 will cause a linking error (i.e., `undefined reference`). Figure 2.57 shows the error message of building `test_case.cpp`. Therefore, CLEMENTINE intentionally does not test inline function without definition.

2.3.20 Templated Function without Definition

Template is a C++ feature that allows the developer to write generic code. Template can be attached with function and class to create templated function and templated class. Then, templated function and templated class can be specialized with various data types. Figure 2.58 shows an example of a templated function without definition. Calling templated function that does not have a definition like in the Figure 2.59 will cause a linking error (i.e., `undefined reference`) Figure 2.60 shows the error message of linking `test_case.cpp`. Therefore, CLEMENTINE intentionally does not test templated function without definition.

Chapter 3. Empirical Evaluation

3.1 Experiment Setup

3.1.1 Research Question

The research questions for this thesis are as follows:

RQ1: How is the general applicability of CLEMENTINE compared to CITRUS? To what extent CLEMENTINE are able to generate test cases for real-world C++ programs in various domains? I applied CITRUS and CLEMENTINE on 8 real-world C++ programs in various domains. Table 3.1 shows 8 real-world C++ programs used to answer this research question.

RQ2: How effective is CLEMENTINE compared to CITRUS in terms of test coverage To what extent does CLEMENTINE achieve code coverage on real-world C++ programs compared to CITRUS? I applied CLEMENTINE for 3 hours on test case generation and performed fuzzing using `libfuzzer` for 1 minute for each test case generated. Then, I compared the code coverage result obtained by CLEMENTINE and CITRUS. Also, I compared the code coverage result reported in the CITRUS's paper [1].

3.1.2 Target Subjects

To show the general applicability of CLEMENTINE (i.e., RQ1), I applied CLEMENTINE and CITRUS on 8 real-world C++ programs that have different domains. Those 8 real-world C++ programs are shown in Table 3.1. `clip` is a command-line program for creating charts and illustrations from a given data. `Exiv2` is a library and command-line program to read, write, delete, and modify image metadata such as Exif, IPTC, XMP, and ICC. `gflags` is a C++ library for command-line flags processing `glog` is an implementation of the Google logging module in C++ programming language. `guetzli` is a JPEG encoder that provides high visual-quality compression for an image. `PcapPlusPlus` is a C++ library whose purpose is capturing, parsing, and crafting network packets. `SQL-parser` is a C++ library that parses SQL query into C++ objects. `Xpdf` is a PDF utility program that can be used to view a PDF file, extract text from a PDF file, convert a PDF file into HTML, and many more. `gflags` and `glog` were selected because those 2 are famous C++ libraries [1]. The other target programs were selected because they have been extensively tested in OSSFuzz [44] or other software testing papers [19].

I also applied CLEMENTINE on 8 target programs used in CITRUS's paper [1]. Table 3.2 shows the target programs and their size in terms of lines of code (LoC). These target programs are used to answer RQ2, that is comparing the testing performance of CLEMENTINE and CITRUS in terms of test coverage.

[1]`gflags` has 2.6k stars and glog 6.1K stars in Github.

Table 3.1: Target Subjects in CLEMENTINE Experiment

Name	Size (LoC)	Commit Hash / Version	URL
clip	17,600	5fca358e	github.com/asmuth/clip.git
exiv2	83,011	ad5484c / 0.27.5	github.com/Exiv2/exiv2.git
gflags	3,954	986e8ee	github.com/gflags/gflags.git
glog	9,510	c525e1a	github.com/google/glog.git
guetzli	8,029	214f2bb	github.com/google/guetzli.git
pcapplusplus	64,805	4b1d0554	github.com/seladb/PcapPlusPlus.git
sql-parser	13,332	44f66fd	github.com/hyrise/sql-parser.git
xpdf	125,529	4.0.3	dl.xpdfreader.com/xpdf-4.03.tar.gz

Table 3.2: Target Subjects Used in CITRUS [1] Experiment

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

Table 3.3: Experiment Setup Information for Subject in Table 3.1

Name	# of preprocessed file	Timeout per function (second)	Time Taken (hour)	Target File of CITRUS
`clip`	81	3	3	`cli.cc`
`exiv2`	78	3	11	`exiv2.cpp`
`gflags`	3	52	3	`gflags.cc`
`glog`	6	14	3	`logging.cc`
`guetzli`	21	21	3	`guetzli.cc`
`pcapplusplus`	79	3	8	`PcapFilter.cpp`
`sql-parser`	10	42	3	`SQLParser.cpp`
`xpdf`	91	3	18	`pdftohtml.cc`

```
vector<ObjCreator> cls_obj_creators = class_type.getObjectCreators();
int idx = RandInt((int) cls_obj_creators.size());
ObjCreator selected_creator = cls_obj_creators[idx]; // CITRUS CRASH HERE
```

Figure 3.1: Pseudocode of CITRUS's crash location

3.2 Experiment Results

3.2.1 RQ1: How is the general applicability of CLEMENTINE compared to CITRUS?

To answer this research question, I applied CLEMENTINE and CITRUS to eight target programs in Table 3.1. This experiment is performed on SWTV-Lab's server that is powered by Intel Core i5-10600 CPU (3.3GHz), 16 GB of RAM, and running Ubuntu 18.04 LTS 64-bit versions. For this experiment, I run CLEMENTINE and CITRUS for only one repetition.

Both CLEMENTINE and CITRUS can only take one input source code file at each run. However, all the target programs in Table 3.1 consist of multiple source code files. Thus for the CLEMENTINE experiment, I run CLEMENTINE on each preprocessed file generated after building the target program, except the test file [2] or example file [3]. I set the timeout for each preprocessed file depending on the number of functions defined in each file. Initially, I set the timeout for 3 seconds for all target programs. However, for some programs that have only a few functions, the total time taken is only a few minutes (e.g., 3 seconds per function for `sql-parser` takes only 12 minutes in total). Therefore, I increase the timeout per function for some target programs so that the total time taken is at least 3 hours. For the CITRUS experiment, I run CITRUS on one file that represents the target program the most. The timeout for the CITRUS experiment is the same as the total time taken for each subject. Table 3.3 shows the number of preprocessed files found in each target, timeout per function in seconds, total time taken in hours, and target file of CITRUS experiment.

Table 3.4 shows the result obtained by CLEMENTINE and CITRUS. CITRUS failed to generate

[2]Source code that used to test the target program, such as unit-test
[3]Source code that contains an example of how to use the target program

Table 3.4: Comparison of Coverage Achieved by CLEMENTINE and CITRUS on Subject in Table 3.1

Subject	**% Statement Coverage**	
	`CLEMENTINE`	`CITRUS`
`clip`	**37.9**	UNCOMPILABLE
`exiv2`	**29.7**	CRASH
`gflags`	48.7	**49.0**
`glog`	**65.6**	53.7
`guetzli`	**39.0**	9.6
`pcapplusplus`	**58.4**	CRASH
`sql-parser`	**34.4**	29.7
`xpdf`	**11.9**	CRASH
Average	**40.6**	35.5

Subject	**% Branch Coverage**	
	`CLEMENTINE`	`CITRUS`
`clip`	**15.3**	UNCOMPILABLE
`exiv2`	**8.4**	CRASH
`gflags`	30.5	**33.4**
`glog`	**39.6**	32.7
`guetzli`	**24.9**	5.4
`pcapplusplus`	**25.5**	CRASH
`sql-parser`	**11.8**	8.2
`xpdf`	**5.5**	CRASH
Average	**20.1**	19.9

Subject	**% Function Coverage**	
	`CLEMENTINE`	`CITRUS`
`clip`	**59.1**	UNCOMPILABLE
`exiv2`	**38.3**	CRASH
`gflags`	**79.4**	77.0
`glog`	**76.8**	68.4
`guetzli`	**63.3**	15.3
`pcapplusplus`	**72.4**	CRASH
`sql-parser`	**71.0**	65.7
`xpdf`	**34.9**	CRASH
Average	**61.5**	56.6

Table 3.5: Statistics of Generated Test Case by CLEMENTINE (CLE) and CITRUS (CIT) on Subject in Table 3.1

Subject	# of Total Attempts		# of Effective TC		# of Uncompilable TC		# of Unlinkable TC		# of Unique Crash TC	
	CIT	CLE	CIT	CLE	CIT	CLE	CIT	CLE	CIT	CLE
clip	7014	1553	0	824	7014	353	0	72	0	14
exiv2	-	6571	-	2796	-	1803	-	406	-	132
gflags	6499	4173	53	181	239	359	319	58	40	34
glog	6560	7132	72	607	516	1445	45	72	55	118
guetzli	9148	4643	29	429	1551	1166	0	0	11	56
pcapplusplus	-	8629	-	3042	-	2015	-	682	-	115
sql-parser	7400	2566	56	315	876	287	0	0	0	79
xpdf	-	64144	-	2954	-	7703	-	45592	-	51

effective test case [4] for four target programs (i.e., `clip`, `exiv2`, `pcapplusplus`, and `xpdf`). For `clip`, all the test cases generated by CITRUS are uncompilable due to the first limitation explained in Subsection 2.1.1 (i.e., method redefinition problem occurred due to including all header files in the test case file). While for the other target programs (i.e., `exiv2`, `pcapplusplus`, and `xpdf`), CITRUS crash due to an internal bug in the implementation. CITRUS removes all unsatisfiable functions (i.e., the function whose arguments can not be resolved by CITRUS) from the target function. However, due to an internal bug, CITRUS failed to remove some unsatisfiable functions and tried to generate a test case to test such functions. Thus, CITRUS crashed while trying to resolve arguments for such functions. To be specific, CITRUS crashed when selecting the "object creator" function to construct an instance of a class whose "object creator" function is not detected. Figure 3.1 shows the pseudocode of location where CITRUS crash. At line 1, CITRUS get the list of "object creator" of a class. However, CITRUS failed to detect the "object creator" of the required class, thus the size of list `cls_obj_creators` is zero (i.e., empty list). Then, `segmentation-fault` error occurred at line 3 because CITRUS tried to access the element of an empty list (i.e., `cls_obj_creators`) using index.

On the other hand, CLEMENTINE success to generate effective test cases for all the target programs. This shows that CLEMENTINE's applicability is better than CITRUS since CLEMENTINE can generate effective test cases for four subjects that CITRUS failed. Moreover, CLEMENTINE also achieves higher function coverage in 75% (3/4) of the target programs (i.e., `glog`, `guetzli`, and `sql-parser`) proving CLEMENTINE has better testing performance compared to CITRUS.

Answer to RQ1: On the eight real-world C++ programs with different domains, CLEMENTINE is able to generate effective test cases for all target programs while CITRUS is able to generate effective test cases for only four target programs.

Table 3.5 shows the number of test cases generated by CLEMENTINE (i.e., "CLE" in the table) and CITRUS (i.e., "CIT" in the table) on eight target programs used to answer RQ1. The column "number of total attempts" means the total number of test cases generated by CLEMENTINE and CITRUS. Note that this number includes test cases that are discarded due to not increasing the coverage or duplicate

[4]Test case that increase the coverage

```
A1: int main () {
A2:  int int0 = 13;
A3:  /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
A4:  google::base::Logger* logger1 = google::base::GetLogger(int0);
A5:  google::base::SetLogger(-88, logger1);
A6:  return 0;
A7: }
```

```
B1: base::Logger* base::GetLogger(LogSeverity severity) {
B2:   MutexLock l(&log_mutex);
B3:   return LogDestination::log_destination(severity)->logger_;
B4: }
```

```
C1: inline LogDestination* LogDestination::log_destination(LogSeverity severity) {
C2:   assert(severity >=0 && severity < NUM_SEVERITIES);
C3:   if (!log_destinations_[severity]) {
C4:     log_destinations_[severity] = new LogDestination(severity, NULL);
C5:   }
C6:   return log_destinations_[severity];
C7: }
```

```
D1: typedef int LogSeverity;
D2: const int GLOG_INFO = 0, GLOG_WARNING = 1, GLOG_ERROR = 2, GLOG_FATAL = 3,
    NUM_SEVERITIES = 4;
```

Figure 3.2: Example of the crash found in `glog`

crashes. Thus, the total number of attempts does not equal the sum of the other columns. The number of effective test cases means the number of test cases that increases the coverage.

Based on Table 3.5, CLEMENTINE is able to generate more effective test cases compared to CITRUS. It also shows that CLEMENTINE found more unique crashes than CITRUS. Although, those unique crashes should be analyzed in more detail to check whether those crashes are true alarms or false alarms.

Figure 3.2 shows an example of crashes found in `glog`. The crashes occurred when calling function `GetLogger` at line A4. The generated test case gives an integer variable (i.e., `int0`) with value `13` to call function `GetLogger`. Then, function `GetLogger` calls function `log_destination` at line B3 and gives `int0` to function `log_destination`. Function `log_destination` has an assertion at line C2 to make sure that the given argument (i.e., `severity`) is within the allowed range that is equal or bigger than zero (i.e. 0) and should be less than `NUM_SEVERITIES`(i.e., the value of `NUM_SEVERITIES` is 4, it is initialized at line D2). However, function `GetLogger` passes `int0` to function `log_destination` and the value of `int0` is 13 which is bigger than 4 (i.e., `NUM_SEVERITIES`). Thus, this crash is caused by assertion failure because the precondition of `severity` is not satisfied.

3.2.2 RQ2: How effective is CLEMENTINE compared to CITRUS in terms of test coverage

To answer this research question, I applied CLEMENTINE and CITRUS to eight target programs used in the CITRUS's paper[1], showed in Table 3.2. This experiment is performed on SWTV-Lab's server that is powered by AMD Ryzen 7 3800XT CPU, 32 GB of RAM, and running Ubuntu 18.04 LTS

Table 3.6: Total time taken for CLEMENTINE and CITRUS

Subject	$t_{\texttt{total}}$ (h)		
	$\texttt{CLEMENTINE}_{3+\texttt{LF1}}$	$\texttt{CITRUS}_{3+\texttt{LF1}}$	$\texttt{CITRUS}_{12+\texttt{LF2}}$ (reported in [1])
`hjson`	12.3	5.0	16.2
`jsonbox`	6.4	4.9	20.5
`jsoncpp`	10.6	6.2	16.8
`json-voorhees`	17.8	5.6	19.3
`jvar`	14.0	5.4	21.7
`re2`	16.8	6.0	20.8
`tinyxml2`	5.3	4.0	17.2
`yaml-cpp`	18.7	4.4	22.7
Average	12.7	5.2	19.4

64-bit versions. For this experiment, I run CLEMENTINE for four repetitions and report the averaged result and standard deviations of the experiments.

Similar to RQ1, I also run CLEMENTINE on each preprocessed file generated after building the target program. However, I set the total timeout to 3 hours for each subject and divide the timeout for each preprocessed file depending on the number of functions. For example, target program `X` consists of file `A.ii`, file `B.ii`, and file `C.ii` which contains 10 functions, 20 functions, and 30 functions respectively. Therefore for the 1-hour experiment, CLEMENTINE will run on file `A.ii` for 10 minutes, file `B.ii` for 20 minutes, and file `C.ii` for 30 minutes. After the 3 hours of test case generation is done, I run `libfuzzer` for 1 minute for each test case. For the CITRUS experiment, I use the same configuration as explained in the CITRUS paper [31] (i.e., create file "`all.cpp`" that includes all header files and target that file). Finally, I compare the coverage result reported in the CITRUS's paper [1]. Table 3.6 shows the total time taken for CLEMENTINE and CITRUS for the 3 hours test case generation and 1 minute libfuzzer. The total time taken for CLEMENTINE is longer than CITRUS because within the 3-hour test case generation, CLEMENTINE generates a larger number of test cases than CITRUS.

Table 3.7 shows the result obtained by CLEMENTINE and CITRUS. CLEMENTINE achieved 85% function coverage or higher on 87.5% (=7/8) of all target programs (i.e., all target programs except `json-voorhees`). `hjson` has the highest improvement in function coverage (69.3%p, from 26.2% to 95.5%) while `tinyxml2` has the higest improvement for statement coverage (51.4%, from 33.9% to 85.3%). On average, CLEMENTINE achieves higher test coverage by 24.4%p (=88.5%-64.1%) for function coverage, 8.9%p (=60.1%-51.2%) for branch coverage, and 15.0%p (=81.6%-66.6%) for statement coverage compared to CLEMENTINE, for the 3 hours test case generation and 1 minute `libfuzzer` experiment. Even compared to the best experiment result reported in CITRUS's paper [1] which has a longer timeout (i.e., $\texttt{CITRUS}_{12+\texttt{LF2}}$ performs 12 hours test case generation and 2 minutes `libfuzzer`), $\texttt{CLEMENTINE}_{3+\texttt{LF1}}$ still achieved higher 13.1%p(=88.5%-75.4%) function coverage and 1.0%p(=81.3%-80.6%) statement coverage. However, Note that the $\texttt{CITRUS}_{12+\texttt{LF2}}$ experiment is performed on the server with different specifications (i.e., Intel Core i5-4670 CPU (3.4GHz), 16 GB of RAM, and running Ubuntu 16.04 LTS 64-bit versions.)

Table 3.7: Comparison of Coverage Achieved by CLEMENTINE and CITRUS on Subject in Table 3.2

Subject	**% Statement Coverage (± stdev.)**		
	CLEMENTINE$_{3+\text{LF1}}$	CITRUS$_{3+\text{LF1}}$	CITRUS$_{12+\text{LF2}}$ (reported in [1])
hjson	79.1 (±2.1)	67.3 (±1.5)	**80.2** (±1.1)
jsonbox	**94.3** (±1.5)	89.6 (±2.2)	93.9 (±1.3)
jsoncpp	63.3 (±1.0)	57.9 (±0.7)	**95.4** (±0.1)
json-voorhees	**77.2** (±0.8)	63.9 (±4.8)	76.7 (±1.0)
jvar	**88.1** (±0.6)	64.5 (±1.0)	81.2 (±2.5)
re2	79.6 (±1.1)	78.5 (±1.2)	**80.2** (±1.0)
tinyxml2	**85.3** (±1.2)	33.9 (±1.3)	56.6 (±3.1)
yaml-cpp	**85.9** (±1.8)	77.3 (±3.0)	80.6 (±0.8)
Average	**81.6**	66.6	80.6

Subject	**% Branch Coverage (± stdev.)**		
	CLEMENTINE$_{3+\text{LF1}}$	CITRUS$_{3+\text{LF1}}$	CITRUS$_{12+\text{LF2}}$ (reported in [1])
hjson	64.6 (±2.2)	58.5 (±2.1)	**70.2** (±1.6)
jsonbox	77.0 (±2.7)	78.1 (±3.0)	**78.9** (±2.2)
jsoncpp	42.1 (±1.6)	45.9 (±0.3)	**60.7** (±0.2)
json-voorhees	47.3 (±1.3)	38.9 (±6.6)	**48.3** (±1.3)
jvar	63.9 (±0.7)	44.2 (±1.2)	**64.5** (±3.8)
re2	59.8 (±1.0)	**59.9** (±1.1)	45.5 (±4.3)
tinyxml2	61.7 (±1.5)	23.8 (±1.2)	**62.4** (±0.9)
yaml-cpp	**64.4** (±3.2)	60.1 (±2.2)	63.0 (±1.0)
Average	60.1	51.2	**61.7**

Subject	**% Function Coverage (± stdev.)**		
	CLEMENTINE$_{3+\text{LF1}}$	CITRUS$_{3+\text{LF1}}$	CITRUS$_{12+\text{LF2}}$ (reported in [1])
hjson	**95.5** (±0.4)	26.2 (±0.4)	38.1 (±0.5)
jsonbox	**96.9** (±0.5)	88.7 (±1.8)	92.6 (±1.5)
jsoncpp	85.7 (±1.1)	68.5 (±0.5)	**95.0** (±0.1)
json-voorhees	**64.5** (±1.5)	55.0 (±1.7)	64.3 (±0.5)
jvar	**95.0** (±0.9)	72.8 (±1.2)	87.0 (±2.4)
re2	**87.8** (±0.4)	84.1 (±2.9)	61.2 (±1.5)
tinyxml2	**90.0** (±0.9)	36.6 (±0.4)	84.2 (±0.9)
yaml-cpp	**92.8** (±0.3)	80.6 (±1.8)	80.8 (±0.7)
Average	**88.5**	64.1	75.4

Table 3.8: Statistics of Generated Test Case by CLEMENTINE (CLE) and CITRUS (CIT) on Subject in Table 3.2

Subject	Avg. # of Total Attempts		Avg. # of Effective TC		Avg. # of Uncompilable TC		Avg. # of Unlinkable TC		Avg. # of Unique Crash TC	
	CIT	CLE	CIT	CLE	CIT	CLE	CIT	CLE	CIT	CLE
hjson	27276.50	3507.75	120.25	558.25	7962.00	551.00	0.00	0.00	1.00	12.25
JsonBox	17044.50	6078.50	113.00	202.00	4261.75	463.75	0.00	0.00	3.50	10.00
jsoncpp	13733.75	2604.75	185.00	454.00	3601.25	369.50	0.00	0.00	8.50	41.00
json-voorhees	10824.25	5799.00	147.25	888.00	3110.50	3240.75	0.00	0.00	4.25	10.75
jvar	13609.50	6798.25	143.25	659.75	3477.00	1715.25	0.00	0.00	28.75	52.00
re2	9037.25	3781.25	179.75	829.25	2105.00	612.00	31.25	73.75	54.75	160.75
tinyxml2	47481.25	13372.25	58.50	137.75	14021.50	2788.25	0.00	0.00	7.00	34.50
yaml-cpp	7374.75	5429.00	77.75	942.00	1863.00	1981.25	3.00	502.50	4.00	27.00

Answer to RQ2: On average, CLEMENTINE achieves higher test coverage than CITRUS by 24.5%p for function coverage, 8.9%p branch coverage, and 15.0%p for statement coverage, for the 3 hours test case generation and 1 minute `libfuzzer` experiment on 8 real-world C++ programs.

Table 3.8 shows the number of test cases generated by CLEMENTINE (i.e., "CLE" in the table) and CITRUS (i.e., "CIT" in the table) on eight target programs used to answer RQ2. Based on Table 3.8, CLEMENTINE generated more effective test cases compared to CITRUS even though CITRUS has a higher number of total attempts than CLEMENTINE. CLEMENTINE also has fewer uncompilable test cases and can find more crashes than CITRUS. However, the crashes found by CLEMENTINE should be analyzed more to check whether the crashes are true alarms or false alarms.

Figure 3.3 shows an example of crashes found in `yamlcpp`. The generated test case creates an instance of `AnchorDict` (i.e., `anchordict0`) at line A3. Then, the generated test case calls function `Get` on `anchordict0` with `122` as the function argument. In function `Get`, the integer `122` is used as the index to access the vector `m_data`. However, the size of vector `m_data` in `anchordict0` is zero since `anchordict0` is just created at line A2. Thus, this crash is caused by accessing an array with an invalid index.

```cpp
int main () {
 YAML::AnchorDict<int> anchordict0{};
 /* PROGRAM CRASHED AT THE EXACT LINE BELOW */
 int int1 = anchordict0.Get((int) 122);
 return 0;
}
```

```cpp
template <class T>
class AnchorDict {
 public:
  AnchorDict() : m_data{} {}
  void Register(anchor_t anchor, T value) {
    if (anchor > m_data.size()) {
      m_data.resize(anchor);
    }
    m_data[anchor - 1] = value;
  }
  T Get(anchor_t anchor) const { return m_data[anchor - 1]; }
 private:
  std::vector<T> m_data;
};
```

Figure 3.3: Example of the crash found in `yaml-cpp`

Chapter 4. Related Works

4.1 C/C++ Unit-level Testing Tools

As the name suggests, unit-level testing is a testing process that tests functions in the program individually. Unit-level testing needs test drivers to test each function individually. There have been many C++ unit testing frameworks such as Google Test [45] and CppUnit [46] that helps the developer to execute test driver automatically. Unfortunately, those frameworks are not capable of generating test drivers automatically. While writing test drivers for small programs may not be difficult, writing test drivers for large programs may take much time and cost.

The performance of unit-level testing relies on how good the test drivers are. The test driver consists of a sequence of functions to provide realistic context for the target function and input value to those function arguments. Coverage-guided greybox fuzzing (e.g., AFL++ [16], `libfuzzer` [18], POWER [19]) and symbolic executions (e.g., CUTE [20], KLEE [21], DeepState [22]) considered as the current state-of-the-art of automated testing technique for C++ programs that can generates various input value. However, those techniques are mainly performed in system-level. Thus, there is still a need for human effort to write test drivers for each function to adapt those techniques in unit-level.

Some early works on automated unit testing utilize static drivers to achieve high test performance in unit testing. KLOVER [23] performs unit testing by generating input using its own C++ symbolic engine on the static drivers. FSX [24] proposed test driver refinement on an existing test driver guided by its *diagnostic engine*. While KLOVER and FSX rely on the static test driver, CLEMENTINE uses random method call sequencce generation to generate test drivers. One positive aspect of random method call sequence compared to static drivers is it can produce diverse object states that help CLEMENTINE to test diverse behavior of the target program.

The later work on automated unit testing focuses on generating fuzz drivers to achieve high testing performance. FUDGE [25] is one of example such work. FUDGE focuses on testing the C++ library and it needs the consumer of the C++ library to generate the test driver. First, FUDGE scans the consumer of the target library to get the candidate target function. Then, FUDGE generates fuzz drivers for each target function by referring to the order of the method call (called *snippet*) that was used in the consumer program. Similar to FUDGE, FuzzGen [26] also focuses on testing C++ library and it requires the consumer code of the target C++ library. FuzzGen utilizes an *Abstract API Dependence Graph* (AADG) of existing external projects to generate the fuzz driver. The difference between FUDGE and FuzzGen relies on how they utilize the consumer of the target library. While FUDGE *sliced* the snippet from the consumer code, FuzzGen extracts the possible method call sequence from AADG to eliminate irrelevant consumer code in the generated fuzz drivers. Although utilizing consumer code can help FUDGE and FuzzGen to generate fuzz drivers, it also has some limitations such as: (1) FUDGE and FuzzGen miss the opportunity to test functions that are never used in the consumer program, and (2) The order of function call sequence is limited to the snippet that exists in the consumer code, thus FUDGE and FuzzGen may not thoroughly explore diverse function call order. On the opposite side, CLEMENTINE does not rely on the consumer code to generate test drivers. Additionally, the random method call sequence generation enables CLEMENTINE to explore diverse function call orders.

IntelliGen [27] is one of the recent works and similar to CLEMENTINE, it also does not need consumer code to generate a test driver. To improve the bug detection performance, IntelliGen generates fuzz drivers for functions in the target C++ library by prioritizing functions with the most potentially vulnerable statements. UTBotCPP [28] is also another recent work that does not need consumer code to generate fuzz drivers. UTBotCPP generates a fuzz driver for each function of the given C++ code and then uses KLEE [21] to generate the input. However, IntelliGen does not explain how it handles C++ features (e.g., template, STL type) and I cannot check whether IntelliGen handles C++ features or not since its implementation is not publicly available. Similarly, UTBotCPP also still does not handle C++ features and it does not generate fuzz drivers for the function that uses C++ feature.

4.2 Method Call Sequence Generation

In programming languages other than C++, method call sequence generation has been widely used to generate test drivers. EvoSuite [10] is the state-of-the-art of automated unit-level testing in Java programming language. Similar to CLEMENTINE, EveSuite performs random method call sequence generation of object-constructing statements to create a test driver. Then, EvoSuite uses genetic evolutionary algorithm to maximize the coverage (e.g., line, branch) and minimize the size of the test cases. To minimize the generated test cases, EvoSuite performs minimization in two different level. The first minimization level is done in test suite level by removing duplicate test cases. The second minimization level is done in test case level by removing duplicate statements. Two benefits of test case minimization are increasing the readability of the test case and reducing the risk of generating flaky tests [47]. EvoSuite is developed as an open-source tool and there have been many improvements since it was first developed in 2011.

There is another automated unit-level testing for Java programs called Randoop [2]. Randoop also uses random method call sequence generation to create the test cases. Once, the test case is generated, Randoop will execute the test case and check for contract validations using *contract checker*. The result of the execution determines whether the test case is redundant, illegal, contract-violating, or useful for generating more test cases. Contract-violating test cases are considered potential errors that should be fixed. For the non-violating test cases, Randoop checks the test cases against given filters to determine whether the test case is valuable or not. There are two main differences between CLEMENTINE and Randoop. The first difference is the way CLEMENTINE and Randoop detect failing test cases. CLEMENTINE considers a test case as fail if the test case crashes while execution, while Randoop uses a contract checker to check whether a test case violates given contracts or not. The second difference is the way CLEMENTINE and Randoop determines whether a test case is valuable or not. CLEMENTINE determines whether a test case is valuable from the test coverage (e.g., line, branch). If the test case increases test coverage, then the test case is considered a valuable test case. On the other hand, Randoop determines a test case is valuable using three defined filters (i.e., equality. null, exception).

Chapter 5. Conclusion

5.1 Conclusions

CLEMENTINE is the next version of CITRUS, an automated C++ unit-level tool that uses method call sequence generation technique to generate high code coverage test suite. CLEMENTINE improves the testing performance of CITRUS by overcoming the limitation of design choices in CITRUS. There are three main limitations of CITRUS addressed in this work, improper way of writing test case file, improper way of linking test case file, and many not-properly-handled functions. Moreover, CLEMENTINE also handles more C++ features like function pointer type, void pointer, non-public member function, and so on.

By solving two main limitations, CLEMENTINE success to generate effective test cases for four target programs which CITRUS failed to do. Adding support to more C++ features also makes the testing performance of CLEMENTINE better compared to CITRUS. On average, CLEMENTINE achieved 81.6% statement coverage (15.0%p higher than CITRUS), 60.1% branch coverage (8.9%p higher than CITRUS), and 88.5% function coverage (24.4%p higher than CITRUS) on eight real-world C++ programs.

5.2 Future Work

CLEMENTINE is able to find more unique crashes on most of the target subjects compared to CITRUS. However, there is still a need to analyze those crashes to check whether those crashes are actually true alarms or are they just false alarms. Thus, analyzing the crashes found by CLEMENTINE is one of the important thing to do in the future.

There are still many things can be done to improve the performance of CLEMENTINE. Implementing support for more C++ types and features surely will improve CLEMENTINE. Although CLEMENTINE already supports some C++ types and features, there are still many C++ types and features that are still not supported by CLEMENTINE. For example, `FILE` type, std::function type, double pointer type, and so on.

Changing the input to multiple files also can improve CLEMENTINE. Currently, CLEMENTINE can take only one input file at a time. Thus, the user should distribute the time budget to each file before running CLEMENTINE and the time budget for each file is static. Although the number of functions can be a proxy to distribute the time budget for each file, the number of functions does not represent the complexity of a file. Thus the time budget distribution might be ineffective.

Improving the mutation operator, especially for random method call insertion. Currently, for random method call insertion, CLEMENTINE randomly selects the inserted method from the list of functions. However, the inserted method may not be related to the target function. Thus inserting not related function may not help CLEMENTINE to explore diverse behavior of the target function. For example, the current test case targets member function `pop` from class `Stack`. Thus, inserting method call to member function `append` from class `Vector` will not help CLEMENTINE to explore diverse behavior of function `pop`.

Bibliography

[1] R. S. Herlim, Y. Kim, and M. Kim, "CITRUS: automated unit testing tool for real-world C++ programs," in *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*, pp. 400–410, IEEE, 2022.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] A. Lee, I. Ariq, Y. Kim, and M. Kim, “POWER: program option-aware fuzzer for high bug detection ability,” in *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*, pp. 220–231, IEEE, 2022.

[20] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C,” pp. 263–272, 2005.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] "UTBotCPP Github's Page." https://github.com/UnitTestBot/UTBotCpp. Accessed: 2023-05-11.

[29] "CMake Website." https://cmake.org/. Accessed: 2023-05-11.

[30] "BEAR Github Page." https://github.com/rizsotto/Bear/. Accessed: 2023-05-11.

[31] R. S. Herlim, *On Understanding and Improving Automated Unit-level Test Generation for C++ Programs*. Daejeon, South Korea: Korea Advanced Institute of Science and Technology, 2022.

[32] 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.

[33] "ar manual page." https://linux.die.net/man/1/ar. Accessed: 2023-05-20.

[34] 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.

[35] "LCOV Modified by henry2cox." https://github.com/henry2cox/lcov/tree/diffcov_initial. Accessed: 2023-05-20.

[36] "C++ Copy Constructor." https://en.cppreference.com/w/cpp/language/copy_constructor. Accessed: 2023-05-17.

[37] "C++ Implicitly Generated Member." https://www.cs.ucf.edu/~leavens/larchc++manual/lcpp_136.html. Accessed: 2023-05-17.

[38] "C++ Abstract Class." https://en.cppreference.com/w/cpp/language/abstract_class. Accessed: 2023-05-17.

[39] "C++ Unnamed Namespaces." https://www.ibm.com/docs/en/i/7.1?topic=only-unnamed-namespaces-c. Accessed: 2023-05-13.

[40] "C++ Member Access." https://www.ibm.com/docs/en/i/7.2?topic=only-member-access-c. Accessed: 2023-05-17.

[41] "Clang: a C language family frontend for LLVM." https://clang.llvm.org/. Accessed: 2023-05-11.

[42] "C++ Operator Overloading." https://en.cppreference.com/w/cpp/language/operators. Accessed: 2023-05-17.

[43] "What is C++ inline functions?." https://cplusplus.com/articles/2LywvCM9/. Accessed: 2023-05-11.

[44] "OSSFuzz Projects List." https://github.com/google/oss-fuzz/tree/master/projects. Accessed: 2023-05-20.

[45] "GoogleTest User's Guide." https://google.github.io/googletest/. Accessed: 2023-05-11.

[46] "CppUnit Test Framework." https://freedesktop.org/wiki/Software/cppunit/. Accessed: 2023-05-11.

[47] 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.

Acknowledgment

First and foremost, I would like to express my deepest gratitude and praise to Allah SWT, the Most Merciful and the Most Gracious, for granting me the strength, wisdom, and perseverance to complete my master's degree. I would also like to acknowledge the profound impact of the teachings and example set by the Prophet Muhammad SAW (peace be upon him). I am also grateful to the Prophet Muhammad (peace be upon him), as His Messenger. His life serves as a constant source of inspiration and moral guidance for millions around the world, including myself

I extend my heartfelt appreciation to Professor Moonzoo Kim, whose expertise, dedication, and unwavering support have been instrumental in shaping the outcome of this thesis. His profound knowledge and insightful guidance have significantly contributed to my academic and personal growth. I am truly honored to have had the privilege of working under his supervision.

I am deeply grateful to my family for their unwavering support, encouragement, and understanding throughout my academic journey. Their love, sacrifices, and belief in my abilities have been the driving force behind my accomplishments. Their constant presence, words of encouragement, and belief in me have provided me with the strength and determination to overcome challenges and reach this milestone.

I would also like to express my sincere gratitude to my fellow labmates: Robert Sebastian Herlim, Ahcheong Lee, Letian Zhang, Youngseok Choi, and Zidong Yang, who have been an incredible source of motivation and inspiration. Their willingness to help and teach me, share knowledge, and provide valuable feedback has greatly enriched my research experience. Their presence has made the lab a nurturing and intellectually stimulating environment. I hope that we can meet again someday in the future and hear about your success and happy stories.

Lastly, I would also like to express my heartfelt appreciation to my friends who have stood by my side, offering support and encouragement throughout this academic journey. Their friendship, words of encouragement, and uplifting spirits have been a source of comfort and motivation.

Curriculum Vitae

Name	:	Irfan Ariq
Birthplace	:	Bandung, Indonesia

Educations

2020. 8. – 2023. 7.	Korea Advanced Institute of Science and Technology (Master Program)
2015. 8. – 2019. 7.	Bandung Institute of Technology (Bachelor Program)
2012. 7. – 2015. 7.	Nurul Fikri Boarding School Islamic Senior High School Serang

Career

2019. 8. – 2020. 7.	NLP Engineer at Prosa AI (Indonesia)

Publications

1. A. Lee, **I. Ariq**, Y. Kim, and M. Kim, " POWER: Program Option-Aware Fuzzer for High Bug Detection Ability", *International Conference on Software Testing, Verification and Validation (ICST)*, 2022, IEEE.