CITRUS: Automated Unit Testing Tool for Real-world C++ Programs

Robert Sebastian Herlim
KAIST
Daejeon, South Korea
robert.herlim@kaist.ac.kr

Yunho Kim
Hanyang University
Seoul, South Korea
yunhokim@hanyang.ac.kr

Moonzoo Kim
KAIST
Daejeon, South Korea
moonzoo.kim@gmail.com

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

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

***Index Terms*—Automated test generation, C++ unit testing, random method sequence generation, code mutation, dynamic analysis**

I. Introduction

C++ programming language is famous for its extensibility, flexibility, and high performance. Thus, it is popular in many application domains that requires significant development efforts such as database engines, operating systems, web browsers, video games, and so on. However, due to the notoriously high complexity of the syntax and semantics of C++, it is technically challenging to develop reliable C++ programs. Even worse, although the need to automatically test C++ programs has been high, there are almost no automated unit testing tool publicly available for real-world C++ programs due to the high complexity of C++ language features (e.g., template instantiation, complex STL types, and so on). Although fuzzing tools [1], [2], [3], [4] generate and mutate input bytes to C++ target programs (without analyzing complex C++ program code), testing effectiveness of such system-level fuzzing is not high within limited time budget because of the huge search space of an entire target program.

To resolve the aforementioned difficulties of C++ programs, we have developed a new automated unit testing tool CITRUS (**C**++ un**I**t **T**esting for **R**eliable and **U**sable **S**oftware) (Section III-B describes how CITRUS handles template instantiation and Section III-C shows how it manages complex STL types). CITRUS receives the source code of a target C++ program P and it automatically generates test driver files for P, each of which consists of various method calls of P. Then, CITRUS generates more diverse test drivers by mutating the test driver code to increase the test coverage of P. In other words, CITRUS explores diverse states of P by executing various method sequences of functions in P. In addition, CITRUS improves the test coverage of P further by applying `libfuzzer` [5] to change P's state by mutating arguments of the methods.

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

The contributions of CITRUS are as follows:

1) CITRUS is one of the very few tools that can automatically generate test cases to test complex real-world C++ programs in a unit-level. [1] [2]
2) CITRUS generates test cases that achieve high testing effectiveness on real-world C++ programs (Section V-B).
3) We have performed experiments on the eight real-world C++ programs to evaluate the testing effectiveness and efficiency of CITRUS (Section V).

The remaining paper consists of the following sections. Section II describes the design of CITRUS regarding how it generates various test cases. Section III describes the detailed implementation of CITRUS. Section IV describes our experiment setup. Section V shows the experiment results to evaluate the testing effectiveness and efficiency of CITRUS. Section VI reports the case study on how CITRUS detects crashes on real-world C++ programs. Section VII explains related work to CITRUS. Finally, Section VIII summarizes the paper with future work.

[1]CITRUS is available at https://github.com/swtv-kaist/CITRUS.
[2]CITRUS demo video is available at https://youtu.be/rrS8Eg%5fKQh8.

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

A. Overview

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

B. Test Cases Generated by CITRUS

A test case generated by CITRUS has the following characteristics. A CITRUS test case tc is defined as a sequence of method invocation statements and the statements to construct arguments of the method calls.

$$tc \overset{\text{def}}{=} \langle s_1, s_2, ..., s_n \rangle \quad (1)$$

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

1) declare a variable of a primitive type with initialization (e.g., `int intVar1 = 7;`), or
2) invoke a method (e.g., `const ClassA &objA1 = ClassA(intVar1);` or `objA1.method1(objB1);`).

Thus, s_i has the following characteristics:

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

For example, the following CITRUS test case (lines 1–4) executes a target method `method1` on an instance of `ClassA` at the line 4, after constructing an object `objA1` of `ClassA` (lines 1–2) and an argument `objB1` for `method1` (line 3).

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

The statement at the line 3 has `ClassB` as its type and `const` as its type modifier. It also has a variable whose name is '`objB1`'.

C. Process of CITRUS

CITRUS operates in the following three stages as depicted in Figure 1:

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

1) Creating Program Representation

First, CITRUS collects the following information from a target program source files (i.e., `.cpp`) through traversing abstract syntax trees (AST) of a target program:

- Lists of `classes`, `structs`, `enums`, and global functions declared in the target program.
- A list of header files (i.e., `.h`, `.hpp`).

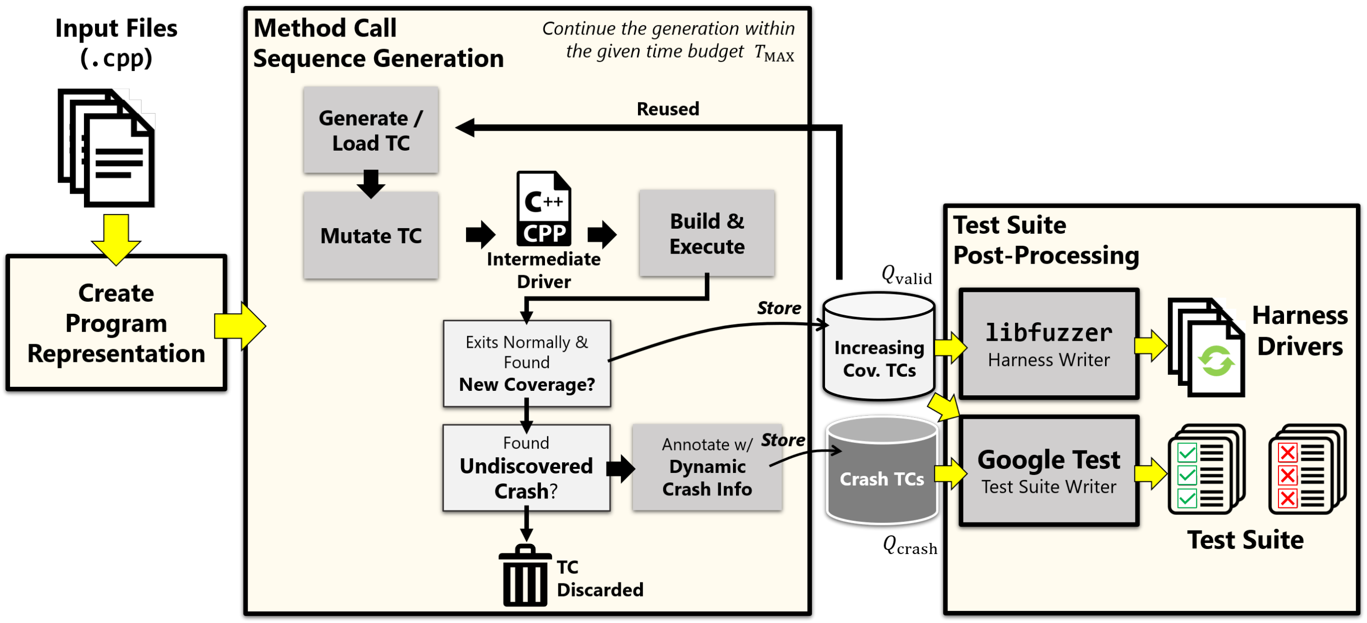
Then, CITRUS mines type information from the information obtained at the AST Traversal stage. CITRUS builds a type system `TS` for `classes`, `structs`, `enums`, and member/global functions of the target program. Algorithm 1 describes how CITRUS builds the type system `TS`. Also, CITRUS constructs an inheritance tree model (ITM) (L3–L6, L16). The ITM supports CITRUS to construct only the relevant type $z \in \{C\} \cup$ `Subclass(C)` while resolving for a class C.

Algorithm 1: Creating Program Representation

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

CITRUS (`RegisterFunc` at L10) distinguishes "object creators" from the regular functions (other method sequence generation techniques [6], [7] apply a similar approach). Any function f that returns a non-primitive type C where $C \notin$ `ArgTypes`(f) is recognized as object creator of class C. Constructors and *static factory* methods are two most-common object creators in object-oriented programming. Also, CITRUS (`RegisterClass` at L7) registers *implicit* object creators for applicable `classes` and `structs`, such as implicitly-declared default constructors [8] and `struct` initialization list [9].

At L17–L19, CITRUS excludes all unsatisfiable functions from the list of functions. We define a function f as an *unsatisfiable* function if there exists a type $t \in$ `ArgTypes`(f), where t is unconstructable by CITRUS (similarly, a method `m` of a class `C` is unsatisfiable if CITRUS cannot construct



Input Files (.cpp)
Create Program Representation
Method Call Sequence Generation
Continue the generation within the given time budget T_{MAX}
Generate / Load TC
Reused
Mutate TC
Intermediate Driver
Build & Execute
Exits Normally & Found New Coverage?
Store
Found Undiscovered Crash?
Annotate w/ Dynamic Crash Info
Store
TC Discarded
Q_{valid}
Increasing Cov. TCs
Crash TCs
Q_{crash}
Test Suite Post-Processing
libfuzzer Harness Writer
Harness Drivers
Google Test Test Suite Writer
Test Suite

Fig. 1: Overview of CITRUS's process

an instance of C). Some examples of unconstructable types: (1) classes with no recognized object creators (e.g., class Y that requires function pointers for construction, which are unsupported by CITRUS) and (2) unhandled STL classes by CITRUS (e.g., thread, mutex, function).

2) Method Call Sequence Generation

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

- If exe's execution terminates normally, CITRUS will store tc into the valid queue Q_{valid} in a case that exe increases coverage.
- Otherwise, CITRUS re-executes exe in gdb environment to collect the crash information such as a stack trace. Then, it puts tc into Q_{crash} if the stack trace has not been generated previously (i.e., a new crash error occurs).

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

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

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

We further elaborate the following core processes of generating test cases in the following subsections:

Algorithm 2: Method Call Sequence Generation

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

Algorithm 3: `LoadOrGenerateTestCase`

```
Input: Type system TS and queue of valid TCs $Q_{valid}$
Output: A candidate test case $tc$ to be executed
1  $b_{gen\_new} \leftarrow$ RandInt$(0, 1)$;            /* 50% prob */
2  if $Q_{valid}$ is empty or $b_{gen\_new} == 0$ then
3      $funcs \leftarrow$ AllFunctions(TS);
4      $f_{target} \leftarrow$ random function selected from $funcs$;
5      $tc \leftarrow$ GenTCForMethod$(f_{target})$;
6  else
7      $tc \leftarrow$ RoundRobinSelection$(Q_{valid})$;
8  end
9  return $tc$
```

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

a) Test Case Generation from Scratch

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

1) an existing statement $s \in stmts$ where `Type`$(s) = type_{arg}$, or
2) constructing another sequence of method calls (and statements to provide primitive arguments) that constructs a new statement s' where `Type`$(s') = type_{arg}$ and appending the sequence to $stmts$.

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

b) Test Case Mutation

Algorithm 5 (`MutateTC`) describes how CITRUS mutates a CITRUS test case. CITRUS ensures every test case mutations to preserve the type validity in the sequence generated.

CITRUS performs three types of test case mutations: insertion, deletion, and modification as described at L5–L10. For statement modification, it performs the following six statement mutation operators as follows (we use the same mnemonic names of mutation in Agrawal et al. [10]): CGCR (Constant Replacement using Global Constant), VLSR (Mutate Scalar References using Local Scalar References), VLTR (Mutate Structure References using only Local Structure References), CLSR (Constant for Scalar Replacement using Local Constants), OAAN (Arithmetic Operator Mutation), and OANG (Arithmetic Operator Negation).

3) Post-processing a CITRUS Test Suite

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

Algorithm 4: `GenTCForMethod`

```
Input: A target function $f$
Output: 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$(stmts)$
```

Algorithm 5: `MutateTC`

```
Input: A CITRUS test case $tc$ to mutate, $MAX$: a
       maximum number of mutations to $tc$
Output: 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 $tc'$
```

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

Additionally, CITRUS outputs the de-duplicated crashing test cases stored in Q_{crash}. Crashing test cases generated by CITRUS are annotated with: (1) gdb stack trace output and (2) a *comment* to point at the crashing line. By these additional information, CITRUS helps a user identify the root cause of the crash (see Section VI for Case Study).

III. Implementation

A. Overview

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

To begin testing, CITRUS requires the following items:

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

The compilation database helps CITRUS extract the compilation flags used for preprocessing and compiling u_{cpp}. However, such compilation databases provide no information about the necessary object files to build an executable file. To mitigate this, the current version of CITRUS requires the user to specify the linking configuration, which covers: (1) the directory where the target program's object files (`.o`) exist, and (2) additional external libraries linking flags (if any, e.g. `-lz` to use the `zlib` library). [3]

CITRUS uses LCOV to measure the testing coverage. To support this, CITRUS requires the target program's binaries to be instrumented for coverage analysis (i.e., compiled using `--coverage` flag). However, due to the C++ exception feature, coverage instrumentation on C++ programs generates too many (almost) unreachable `throw` branches (e.g., during C++ object construction) that are (almost) never executed. CITRUS utilizes a modified version of LCOV [11] to exclude such virtually unreachable branches.

B. Testing C++ Template Classes/Functions

Testing template classes in C++ is challenging because it is almost impossible to instantiate template classes with all possible types. In addition, inappropriate type instantiation of template classes may generate many uncompilable test cases. Listing 1 demonstrates a simple scenario in which inappropriate type instantiation produces test case that cannot be compiled. At L10, `Outer<char>` can be compiled because `Inner(char*)` is defined. Instantiating `Outer<int>` at L11, however, causes a compilation error because `Inner(int*)` is not defined.

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

Listing 1: Challenge in Instantiating Template Classes in C++

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

To reduce the number of uncompilable test cases generated, CITRUS conservatively binds a free type variable `T` to a concrete type according to the existing type bindings in the target program. For example, CITRUS binds `T` to `char` (denoted as $\{$`T` $\mapsto$ `char`$\}$) when it generates a method call sequence for `Decode` at L8 because the argument requires `Outer<char>&` type. However, such type hinting may not always be available, such as when CITRUS targets the constructor of `Outer<T>` at L5. In this case, CITRUS randomly selects either $\{$`T` $\mapsto$ `int`$\}$ or $\{$`T` $\mapsto$ `double`$\}$.

C. Handling C++ STL Classes

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

1) Some STL classes have more *indirect* ways of construction, rather than by simply calling constructors. For example, `unique_ptr` and `tuple` should be constructed through `make_unique` and `make_tuple` API respectively, instead of their constructors.
2) Most STL classes contain many member functions, and including all of such functions in method sequence generation may not contribute to exploring diverse program states. For example, CITRUS may generate `vector` objects (with arbitrary sizes and elements) by using the `push_back` API only. Thus, generating random sequences of method calls with `resize` and `erase` operators on `vector`s is ineffective towards exploring new executions (i.e., just enlarging the search space).

CITRUS mitigates such technical challenges by putting additional engineering efforts to handle the construction of each STL class without performing random sequence generation. When a function f requires an object of a STL class t_{STL} as an argument, CITRUS constructs an *initialized* t_{STL}-typed object o_{STL} by using a *single-line* STL construction statement. For example, CITRUS uses C++ initializer lists [12] to construct objects of STL containers.

Currently, CITRUS handles 23 STL classes in the four categories as follows [4]:

- Containers (e.g., `vector`, `set`, `map`, `forward_list`).

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

TABLE I: Target Subjects

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

TABLE II: Accessible Function Statistics in Target Subjects

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

- Utility (e.g., pair and tuple).
- Strings (e.g., basic_string, string, wstring).
- Memory (e.g., unique_ptr and shared_ptr).

D. Crash De-duplication in CITRUS

Since crash de-duplication task is essential to reduce the effort of the time-consuming manual bug analysis, CITRUS uses stack hashes to triage crashes [13]. To generate stack hashes, CITRUS extracts a sequence of source locations (i.e., file names + line numbers) in the function call stack parsed from the gdb stack trace output. CITRUS uses source code locations (instead of binary code locations) because the binary code locations may be inconsistent among different runs due to constantly changing executable file during method call sequence generation. Note that CITRUS considers only the source locations in the target project directory to avoid duplicate crashes caused by uncontrolled behaviors of external library function calls.

IV. Experiment Setup

A. Research Questions

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

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

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

B. Target Subjects

We applied CITRUS on the eight popular real-world C++ programs in Table I ranging from 1.5KLoC (jsonbox) to 20KLoC (re2). Those programs contain complex C++ language features, such as polymorphism, template classes, STL types, and so on.

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

- For tinyxml2, 158 functions were unsatisfiable because CITRUS failed to recognize the non-static member functions (that returns a particular class) for constructing one of its argument types [5].
- For json-voorhees, 111 functions were due to argument dependency to one of unhandled STL classes (e.g., std::type_index, std::type_info); 83 functions were due to *type aliasing* [14] within template classes, which are still unhandled by CITRUS; and 28 functions were due to dependency with *other* unsatisfiable functions (which had been excluded).
- The remaining unsatisfiable functions (8 in tinyxml2 and 11 in json-voorhees) were mostly caused by unhandled argument types by CITRUS, such as: multi-dimensional pointers, opaque pointers (void*), and FILE.

C. CITRUS Variants

To answer RQ2, we make four CITRUS variants to compare. The four CITRUS variants are as follows:

- $\mathrm{C}_{12+\mathrm{LF}2}$, which generates sequences of method calls for 12 hours and applies libfuzzer for two minutes for each test case generated. This is the main configuration of CITRUS.
- $\mathrm{C}_{6+\mathrm{LF}1}$, which generates sequences of method calls for six hours and applies libfuzzer for one minute for each test case generated.
- $\mathrm{C}_{6+\mathrm{LF}3}$, which generates sequences of method calls for six hours and applies libfuzzer for three minutes for each test case generated.
- C_{24}, which generates sequences of method calls for 24 hours.

[5]We did not utilize such non-static functions as object creators as they do not always perform object construction. For instance, it is common to write a setter method that returns this reference for method chaining, such as: “Point *SetX(int x) { x_ = x; return this; }”

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

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

D. Environment Setup

We conducted our experiments in our cluster in which each node is equipped with Intel Core i5-4670K CPU (3.4 GHz) and 16GB RAM, running Ubuntu 16.04 64-bit version. We reported the result of our experiments collected from ten repeated runs to reduce the random variance caused by the randomized algorithm. We used $MAX = 20$ as the maximum number of mutation operations.

E. Threats to Validity

The possible threat to external validity is the generality of our subject selection. To reduce the risk, our target subjects consist of C++ open-source programs of varying sizes. The threat to internal validity is a bug in CITRUS implementation. To reduce the risk, we carefully wrote and extensively tested our CITRUS implementation.

V. Experiment Results

A. Statistics on CITRUS Test Cases

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

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

The experiment result shows that the test cases generated by CITRUS achieved high statement coverage and branch coverage. Figure 2 shows the statement coverage and the branch coverage obtained by the test cases generated by CITRUS ($\texttt{C}_{12+\texttt{LF2}}$) on the eight target subjects. CITRUS achieved roughly 80% or higher statement coverage in 87.5% (=7/8) of all target subjects (i.e., in all target subjects except `tinyxml2`). For `jsonbox` and `jsoncpp` subjects, it even achieved >90% statement coverage. For branch coverage, CITRUS achieved roughly 60% or higher in the majority of the target subjects (=6/8) (i.e., in all target subjects except `tinyxml2` and `json-voorhees`).

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

*C. **RQ2**: How effective and efficient is CITRUS compared with other CITRUS variants?*

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

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

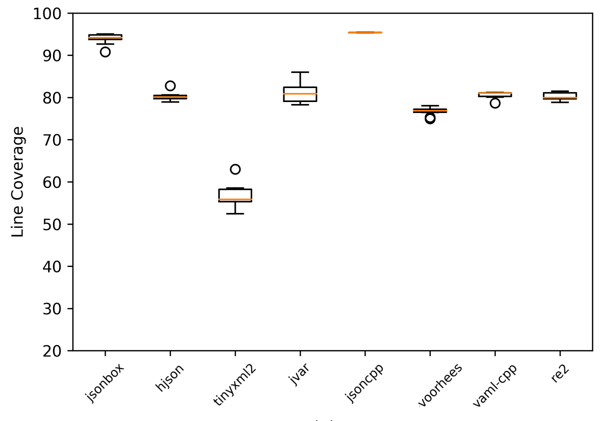
Table V shows the time cost consumed by the four CITRUS variants. All three CITRUS variants that utilize `libfuzzer` spent less time than that of $\texttt{C}_{24}$. In particular, $\texttt{C}_{12+\texttt{LF2}}$ spent 19%(=(24-19.4)/24) less time than $\texttt{C}_{24}$, but achieved 12% higher (=(80.6-72.0)/72.0)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

As a result, from Table IV and Table V, we can confirm that the idea of integrating `libfuzzer` into CITRUS is effective and efficient to increase test coverage. This is because since the CITRUS variants that utilize `libfuzzer` (i.e., $\texttt{C}_{6+\texttt{LF1}}$, $\texttt{C}_{6+\texttt{LF3}}$, and $\texttt{C}_{12+\texttt{LF2}}$) achieved higher statement and branch coverage than $\texttt{C}_{24}$ and the time costs of the CITRUS variants that utilize `libfuzzer` were lower than that of $\texttt{C}_{24}$.

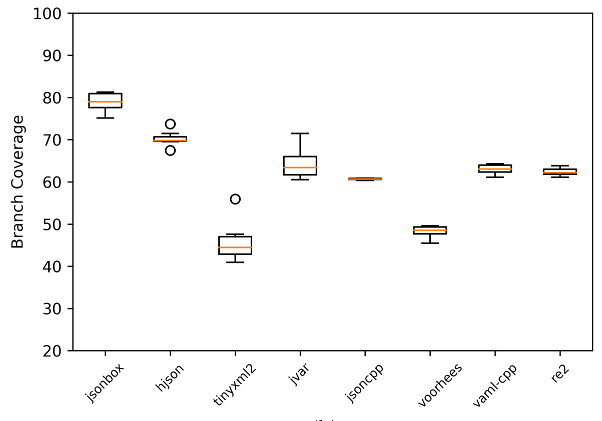
Answer to RQ2: In most subjects, CITRUS variant $\texttt{C}_{12+\texttt{LF2}}$ produced the best result. Also, applying `libfuzzer` helps CITRUS improve the coverage score and time cost, compared to the technique that uses only method call sequence generation ($\texttt{C}_{24}$ variant).

VI. Case Study of Crash Detection

We have conducted a case study to show how CITRUS detects crash bugs in a real-world C++ program. We selected



(a)



(b)

Fig. 2: Statement and Branch Coverage Achieved by CITRUS ($\texttt{C}_{12+\texttt{LF2}}$) on the Target Programs

TABLE IV: Coverage Achieved by CITRUS Variants

Subject	**Avg. Statement Coverage (%)**				**Avg. Branch Coverage (%)**				**Avg. Function Coverage (%)**			
	$\texttt{C}_{24}$	$\texttt{C}_{6+\texttt{LF1}}$	$\texttt{C}_{6+\texttt{LF3}}$	$\texttt{C}_{12+\texttt{LF2}}$	$\texttt{C}_{24}$	$\texttt{C}_{6+\texttt{LF1}}$	$\texttt{C}_{6+\texttt{LF3}}$	$\texttt{C}_{12+\texttt{LF2}}$	$\texttt{C}_{24}$	$\texttt{C}_{6+\texttt{LF1}}$	$\texttt{C}_{6+\texttt{LF3}}$	$\texttt{C}_{12+\texttt{LF2}}$
`jsonbox`	93.6	93.7	**94.2**	93.9	75.8	78.6	**79.1**	78.9	**93.3**	92.6	92.6	92.6
`hjson`	70.2	78.8	79.7	**80.2**	57.6	68.5	69.8	**70.2**	36.8	37.5	37.6	**38.1**
`tinyxml2`	**59.5**	52.9	53.2	56.6	**49.1**	41.5	42.1	45.5	**63.8**	59.0	59.3	61.2
`jvar`	**84.5**	80.6	80.8	81.2	**69.7**	63.7	64.0	64.5	**89.5**	86.9	86.9	87.0
`jsoncpp`	60.3	59.7	60.1	**95.4**	45.3	46.9	47.2	**60.7**	74.2	72.7	72.8	**95.0**
`json-voorhees`	69.3	74.7	75.6	**76.7**	41.8	48.9	**50.3**	48.3	59.9	61.3	61.7	**64.3**
`yaml-cpp`	67.2	78.4	79.0	**80.6**	45.9	60.9	61.6	**63.0**	77.2	79.3	79.5	**80.8**
`re2`	71.1	77.4	79.1	**80.2**	51.3	59.3	61.2	**62.4**	82.0	82.4	83.3	**84.2**
Average	72.0	74.5	75.2	**80.6**	54.6	58.5	59.4	**61.7**	72.1	71.5	71.7	**75.4**

TABLE V: Time Cost for CITRUS Variants

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

five target crash bugs on the eight target subjects with the following criteria:

1) The target crash bug was reported in the `git` commit message that contains one of the following keywords: “crash”, “segmentation fault”, “SIG”, and “SEGV”.
2) The fixed code is still available (i.e., not removed since the patch date) in the latest version of a target program.
3) The crash bug resides in the target program source code.
4) The crash occurs due to internal logic problems, not external environment-related problems (e.g., OS, external files, environment variables, etc.).
5) The crash bug can be understood without deep domain expertise. Otherwise, we are unable to check if a unit-level crashing execution of a CITRUS test case really conforms to that of the reported crash bug. [6]

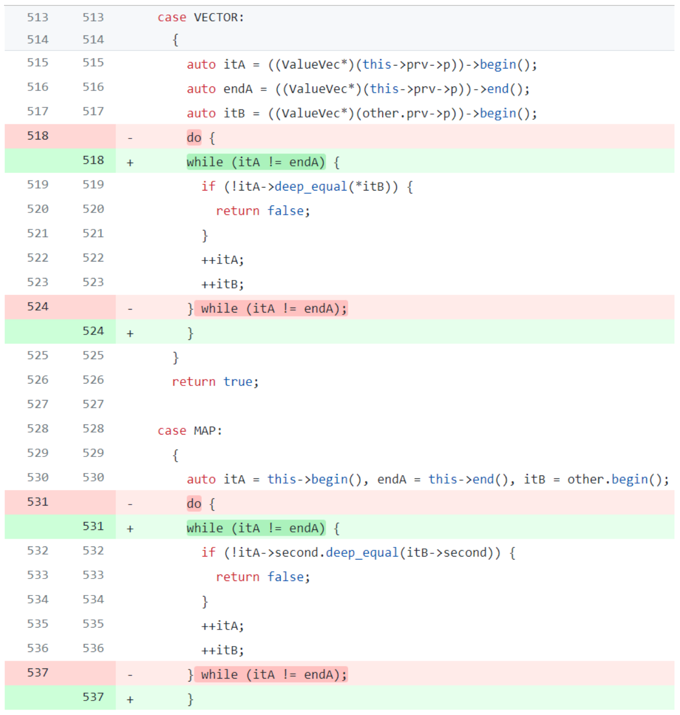
Table VI shows the five target crash bugs selected by the above criteria. For each target crash bug, we manually set up the unit-level crash replication environment to check if a crashing execution of a CITRUS test case really conforms to that of the reported crash bug. We ran CITRUS three times with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

We explain the detail of how CITRUS detects the following crash bug in `hjson` (see the second line in Table VI). The root

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

TABLE VI: Five Target Crash Bugs for the Crash Detection Study

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



```
      case VECTOR:
        {
          auto itA = ((ValueVec*)(this->prv->p))->begin();
          auto endA = ((ValueVec*)(this->prv->p))->end();
          auto itB = ((ValueVec*)(other.prv->p))->begin();
-         do {
+         while (itA != endA) {
            if (!itA->deep_equal(*itB)) {
              return false;
            }
            ++itA;
            ++itB;
-         } while (itA != endA);
+         }
        }
        return true;

      case MAP:
        {
          auto itA = this->begin(), endA = this->end(), itB = other.begin();
-         do {
+         while (itA != endA) {
            if (!itA->second.deep_equal(itB->second)) {
              return false;
            }
            ++itA;
            ++itB;
-         } while (itA != endA);
+         }
```

Fig. 3: Crash Fix in `hjson` `e8f8693` commit

cause of the fix `e8f8693` in `hjson` is an *off-by-one* error in `deep_equal` comparison on empty vectors (or empty maps). As shown in Figure 3, prior to `e8f8693`, `deep_equal` used `do...while` to compare the elements, causing the loop body to be still executed on empty vectors (or maps); the patch avoids the loop body execution for empty vectors (or maps) by changing `do {...} while(c)` to `while(c) {...}`.

Listing 2 shows a crashing CITRUS test case that detects the crash bug in `hjson`. At L17, CITRUS points to the crashing line that invoke `deep_equal`. Since `value5` of `value5.deep_equal(value3)` at L18 is an empty map (see L13 and L16 where `char2` is an empty string ""), the test case triggers the crashing bug and crashes. Also, CITRUS annotates the crashing test case with additional gdb backtrace information (L2–L10)

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

VII. Related Work

A. *C/C++ Unit-level Testing Tools*

Automated unit test case generation for object-oriented programming languages (e.g., C++) is well-known to be a challenging task due to the its large search space. To achieve a high coverage in a target program, testing tools must be able to: (1) generate diverse test harnesses (a.k.a., drivers) to represent realistic contexts to a target unit in the target program; and (2) generate the suitable inputs to increase the test coverage. Most state-of-the-art techniques, such as symbolic executions (e.g., CUTE [15], KLEE [16], DeepState [17]) and coverage-guided fuzzing (e.g., AFL++ [3], `libfuzzer` [5]) require manual efforts by human testers on writing the unit-level test harnesses before starting the input generation. Such manual interventions are indeed costly and ineffective [18] on achieving high test coverage [7]. Even though several techniques (e.g., CONBRIO [19], FOCAL [20], MAIST [21]) have been developed to reduce the manual efforts of generating unit-test harnesses by automatically generating unit test harnesses, they do not support C++ programs because of the high complexity of C++ language features. Meanwhile, in contrast to C++ unit testing frameworks (e.g., Google Test [22], CppUnit [23]) that do not automatically generate test cases (i.e., purposefully only for running unit test cases), CITRUS generates C++ unit tests automatically which can be later incorporated to be run on these C++ testing frameworks.

The later approaches on automated unit testing for C++ programs started to incorporate the harness generation process inside the testing process. KLOVER [24] automatically generates static drivers which is later incorporated with its own C++ symbolic execution engines to generate inputs. FSX [25] introduces incremental driver refinement and relevant input analysis to improve the effectiveness of static drivers in symbolic executions. However, utilization of such static drivers may have limitation in triggering behaviors caused by the different ordering of method calls. CITRUS generates test cases through the random method call sequence generation to execute diverse program behaviors.

Recent techniques focus to synthesize fuzz drivers to achieve high test effectiveness. For example, FUDGE [26] and FuzzGen [27] synthesize fuzz drivers by scanning an

[7]We were not able to perform experiment to achieve apple-to-apple comparison between CITRUS and existing tools (such as DeepState and AFL) because most tools do not automatically generate test drivers, while CITRUS generates the test drivers end-to-end.

Listing 2: CITRUS Test Case that Detects The Crash Reported in `hjson` `e8f8693` commit

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

existing external library consumer project to collect candidate entry functions. CITRUS works in a more flexible way as it does not require any external project to perform the method sequence call generation. IntelliGen [18] synthesizes fuzz drivers for functions with most potential vulnerable statements (e.g., pointer dereferencing). However, we could not check if IntelliGen supports C++ language features since their implementation is not publicly available. CITRUS was developed to test complex C++ programs and CITRUS is publicly available. Moreover, compared to the three driver synthesis techniques mentioned above, CITRUS adopts both generative and mutational strategy to improve the diversity of the test cases generated.

B. Method Call Sequence Generation

Method call sequence-based test case generation has been widely applied to programming languages other than C++, such as Java [6], [28] and Python [29]. EvoSuite [6] constructs test suites through evolutionary algorithm to maximize the coverage goals (e.g., line, branch, weak mutation) while still minimizing the test suite size. Randoop [28] performs random sequence generation while utilizing *contract checker*s to find any violations within the code executions. Meanwhile, Garg et al. [30] ported a C++ version of Randoop but its implementation is not publicly available.

Bach et al. [7] conducted a large-scale survey on seven large C++ projects to confirm the importance of Object Creation Problem (OCP) while writing unit tests. While they have suggested an approach to robustly construct a valid solution for most classes, the survey did not discuss how such method call sequences should be utilized to increase the diversity of object states generated. Meanwhile, CITRUS not only focuses on object creation, but also extensively mutates test cases to construct diverse object states.

VIII. Conclusion and Future Work

This paper presents CITRUS which is a new automated C++ unit-level testing tool to generate random method call sequences to produce a test suite achieving high test coverage. To test complex real-world C++ programs, CITRUS handles challenging technical issues such as template instantiation, complex STL classes, and so on. On the eight real-world C++ target programs, we have demonstrated that CITRUS achieved high statement coverage (up to 95%) and high branch coverage (up to 79%). CITRUS is publicly available at

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

For future works, we plan to enhance our CITRUS implementation to handle more complex C++ features, such as STL classes from the more recent C++ language features. Also, to improve the testing effectiveness and efficiency of CITRUS, we will develop new heuristics that utlize mutant execution results to increase test coverage [31], [32] and to reduce the number of mutants [33]. Finally, we will study a method to identify/generate test oracles for C++ test cases as future work since test oracle problem is also an important feature in test case generation.

Acknowledgments

This work was supported by the NRF grant (NRF-2020R1C1C1013996, NRF-2021R1A2C2009384, and NRF-2021R1A5A1021944) and the IITP grant (no. 2021-0-00905-001) funded by the Korea government (MSIT).

References

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

[2] 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.

[3] 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.

[4] A. Lee, I. Ariq, Y. Kim, and M. Kim, "POWER: Program option-aware fuzzer for high bug detection ability," in *the 15th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2022.

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

[6] 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.

[7] 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.

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

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

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

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

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

[20] Y. Kim, S. Hong, and M. Kim, "Target-driven compositional concolic testing with function summary refinement for effective bug detection," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 16–26, 2019.

[21] Y. Kim, D. Lee, J. Baek, and M. Kim, "Concolic testing for high test coverage and reduced human effort in automotive industry," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, p. 151–160, IEEE Press, 2019.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] Y. Kim, S. Hong, B. Ko, D. L. Phan, and M. Kim, "Invasive Software Testing: Mutating Target Programs to Diversify Test Exploration for High Test Coverage," IEEE 11th International Conference on Software Testing, Verification and Validation, April 2018.

[32] Y. Kim and S. Hong, "DEMINER: test generation for high test coverage through mutant exploration," *Software Testing, Verification and Reliability*, vol. 31, no. 1-2, p. e1715, 2021. e1715 stvr.1715.

[33] Y. Kim and S. Hong, "Learning-based mutant reduction using fine-grained mutation operators," *Software Testing, Verification and Reliability*, p. e1786.