KLEE-Unit: KLEE-Driven Unit Test Generator

Zikai Liu
liuzik@student.ethz.ch
ETH Zurich
Zurich, Switzerland

ABSTRACT

Symbolic execution is a powerful testing technique that has been applied to many real-world programs. However, its scalability is yet an ongoing challenge, where the complexity of the software is the limiting factor. Moving in the other direction, in this work, we discuss an attempt to apply symbolic execution to unit testing. We present KLEE-Unit, a unit test generator driven by KLEE, a state-of-the-art symbolic execution engine. KLEE-Unit works as an interactive tool that helps the user reduce manual effort to write unit tests. It analyzes the target function, generates extendable test drivers, invokes KLEE, gathers the test cases, and selectively translates them to unit tests written in unit testing hardnesses like Catch2. KLEE is extended to support intrinsic functions for unit testing. We perform preliminary evaluation on a simple self-contained function as well as a more complicated module built with CMake. In these two case studies, KLEE-Unit helps reduce the user effort needed to produce the same test cases compared with manual writing. We also discuss the limitations and possible future directions of the system.

1 INTRODUCTION

Symbolic execution is a powerful testing technique that has been proven to be effective in finding program defects. However, the scalability of symbolic execution is still an ongoing challenge. The time spent testing real-world programs with the state-of-the-art symbolic engine like KLEE [1] is often at the scale of hours or even days [3], while the reachable coverage may still be insufficient for practical use in most software development.

The scalability problem typically arises from the complexity of the software. When multiple components are integrated, the number of execution paths can be exponential. This motivates us to think about the other way: applying symbolic execution to unit testing, where the units under test (UUT) tend to be much simpler.

In addition to the scalability, another challenge, for not only symbolic execution but also other automatic testing techniques, is the test oracle problem—how to determine whether the observed behavior is expected or not. Many works of symbolic execution focus on undesired behavior that can be directly identified, such as program crashes or undefined operations [1] [3]. In some specific domains, differential testing is adopted to address the issue [6], but there is no easy and widely applicable solution yet.

We do not intend to address the general test oracle problem in this work. Rather, in the scenario of unit testing, manual assertion of the output is typically feasible, as UUT is relatively simple. Furthermore, if the developer practices test-driven development (TDD) or TDD-like strategies (not necessarily writing test beforehand but closely), with a refreshed understanding of the UUT, the manual effort to verify an output value is relatively small.

In this work, we present KLEE-Unit, a system that closely integrates KLEE to generate unit test cases for C functions. At high level, the system acts as a bridge from UUT to KLEE and back to unit testing code. It generates a test driver for the target function, runs KLEE, and translates them to unit tests written in test harnesses like Catch2 [4]. Primarily evaluation shows that the system can help generate the same or equivalent test cases as manually written by the developer, while reducing the lines of code that the user needs to write. However, due to the limited time and resources, we do not perform a wide-range user study.

The remainder of this paper is organized as follows. In the next section, we describe the implementation of the system and how each of these functionalities may help users generate test cases with less manual effort. In Section 3, we present two examples of how the system can possibly be used. Then in Section 4, we discuss the generalizability and limitations of the system. Finally, Section 5 provides related works and Section 6 offers our conclusions.

2 APPROACH AND IMPLEMENTATION

KLEE-Unit consists of the core module, a graphic user interface (GUI) implemented in Qt [2], and an adapted KLEE. Figure 1 shows the overview of the system. The system is designed to be an interactive tool that developers can use during the development process. The system provides flexibility for the user by providing several templates for the test driver and allowing free editing of the driver function. Currently, the system support Catch2 [4] unit testing harness. In this section, we describe the implementation and usage of the system, in the order of its workflow.

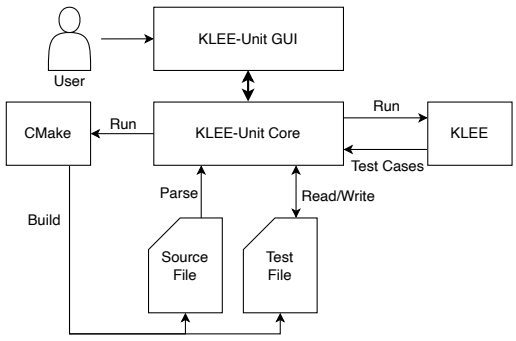

Figure 1: KLEE-Unit overview.

The ease of use is taken into consideration at the very beginning of the development. We design the GUI along with the core module. Figure 2 shows the GUI implemented in Qt [2], on which the user performs the whole workflow. We use the MonacoThe core module is designed as a standalone Python module for flexible adaptations in the future, to an IDE extension for example.

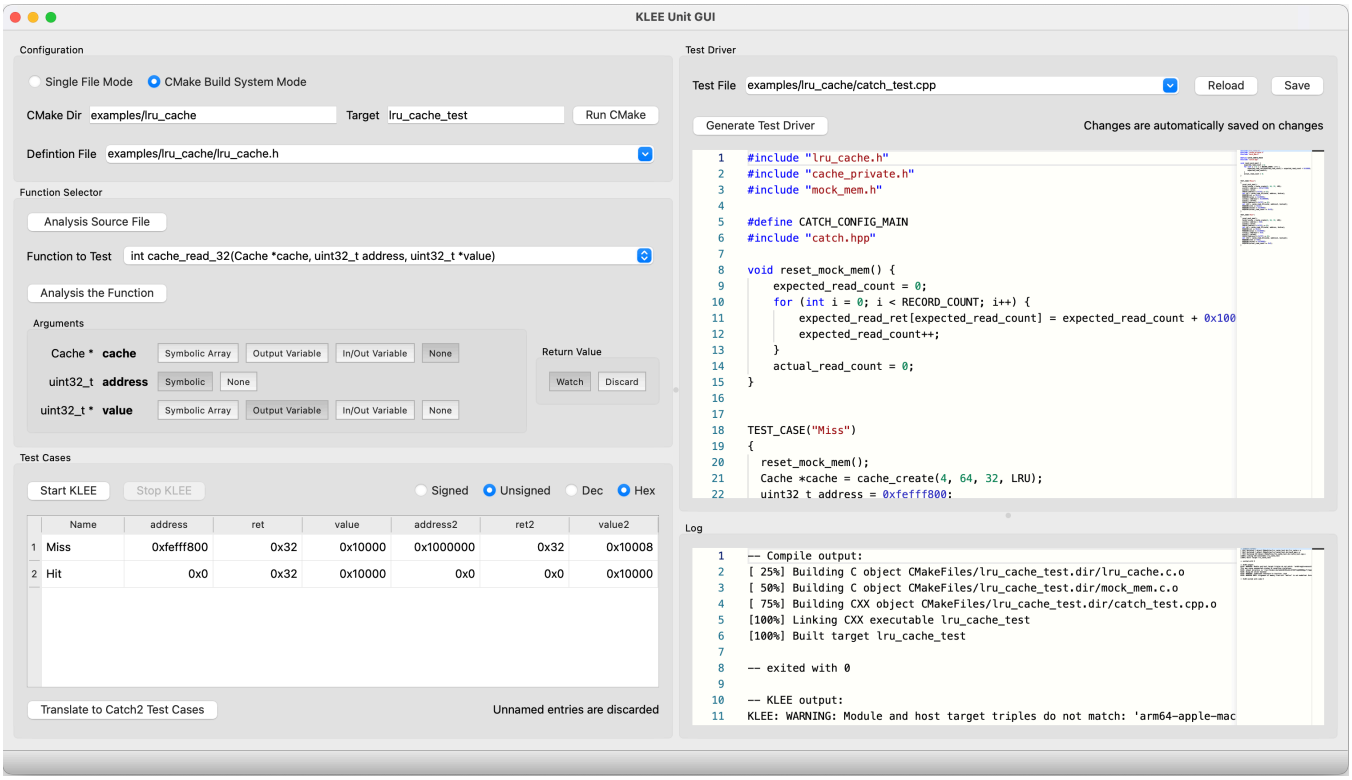

Figure 2: KLEE-Unit GUI implemented in Qt

2.1 Analyze Target Function

The workflow starts by analyzing the signature of the function to test. The system parses the C header or source file where the target function is in, using the pycparser library. The file needs to be processed by the C preprocessor (`cpp`) before it can be parsed by pycparser. We use the stub libc headers shipped with pycparser as well as including several stub headers using the `-include` argument.

pycparser generates an abstract syntax tree of the file. The function declarations (`FuncDecl`s) are gathered into a list. The functions found in the file are shown in the drop-down menu at the top-right of the GUI.

2.2 Configure and Generate Test Driver

KLEE-Unit generates a template driver for the function, which the user can configure and edit before the symbolic execution. The test driver uses a different set of intrinsics from the KLEE for both ease of use and easy translation to concrete test cases. Table 1 shows the list of KLEE-Unit intrinsics. Depending on the type of the parameter, the user can choose among several types of drivers to generate. Table 2 shows the available options for each type of parameter and the templates of each parameter driver. These options are common use cases, while the user is allowed to freely edit the test driver afterward. If the function has a return value, the user can choose to watch the return value or not. We show specific examples in Section 3.

KLEE-Unit targets only C but not C++ code, due to the limited C++ support by KLEE and the parser we are using. However, most unit testing harnesses available for C/C++ require the test cases to be written in C++ for automatic test discovery. We would like to make use of that feature and generate test cases in-place right inside the test file for ease of use. Therefore, KLEE-Unit allows cpp source as the working test file but parsed as a C file.

2.3 Translate and Compile KLEE Driver

After the user edits the test driver, the function is parsed from sketch again and translated to a KLEE driver function, based on the relationship shown in Table 1. We make a few changes in KLEE to accommodate KLEE-Unit's usage.

2.3.1 A New KLEE Intrinsic for Watched Variables. Until KLEE v2.3, only symbolic variables are written to the generated test cases (`ktest` files). However, for the usage of KLEE-Unit, we need values of some non-symbolic result variables for assertions in the unit test. Making a variable as symbolic `klee_make_symbolic` does not work since it only outputs a feasible assignment for the variable at the beginning rather than its final value. Therefore, we design another KLEE intrinsic for this purpose:

```
void klee_watch_obj(void *ptr, const char *name);
```

The function takes the pointer to a variable and a name and asks KLEE to output its value in the specific test case. Notice that KLEE outputs the variable value at the end of execution, rather than at the point where this function is called.

Table 1: Intrinsics used in KLEE-Unit test driver and their translations to KLEE and Catch2 intrinsics.

KLEE-Unit Intrinsics	KLEE Intrinsics	Catch2 Intrinsics
type var = SYMBOLIC(type);	type var; klee_make_symbolic(&var, sizeof(type), "var");	type var = value;
LET(cond);	klee_assume(cond);	CHECK(cond);
WATCH(var);	klee_watch_obj(&var, "var");	REQUIRE(var == value)

Table 2: Available driver template for different types of parameters.

Name	Available for Parameter	Driver Template
None	All	type arg = ?; func(arg);
Symbolic	Types passed by value	type arg = SYMBOLIC(type); func(arg);
Output Variable	Pointer types	type arg; func(&arg);

In/Out Variable	Pointer types	type arg = SYMBOLIC(type);
		func(&arg);
		WATCH(arg);

Expanded Array	Fixed length arrays	type arg[len] = {SYMBOLIC(type), ...}; function(arg);

We add a list of watched variables in KLEE's symbolic state. `klee_watch_obj` registers the variable to the list. When generating test cases, in addition to the symbolic variables, KLEE evaluates the value of the watched variables `after` specific concrete values are selected for all symbolic variables. The solver should return a unique value in this case.

2.3.2 Single File Mode vs Build System Mode. KLEE requires complete knowledge of the test program to perform symbolic execution. Most real-world programs are far more complicated, involving a number of source files and possibly libraries, and are built through build systems. And for unit testing, it is common that the UUT and the test code are put in separate files even if the UUT is contained in one file.

KLEE-Unit provides two modes: the single file mode and the CMake build system mode, which can be selected on the top-right of the GUI (Figure 2). In the single file mode, the source file is directly copied to the test driver file using the `#include` derivative. The test file can be compiled to LLVM bitcode directly with a single call to Clang. This mode is applicable for fast testing simple self-contained components without setting up a build system. In the CMake build system mode, KLEE-Unit works with CMake to build the UUT and the KLEE driver. The user needs to specify the CMake project root directory and target that compiles the UUT and the test driver. KLEE-Unit executes CMake on the project by replacing the C and CXX compilers with `wllvm` and `wllvm++` and including several header files before compiling every translation unit:

```
CC=wllvm CXX=wllvm++ WLLVM_CONFIGURE_ONLY=1 LLVM_COMPILER
    =clang cmake <project directory>  \
-DCMAKE_C_FLAGS='-include catch_dummy.hpp -include
    klee_dummy.h'  \
-DCMAKE_CXX_FLAGS='-include catch_dummy.hpp -include
    klee_dummy.h'
```

The build directory is a temporary directory created as KLEE-Unit starts. When compiling the test driver, KLEE-Unit calls `make` to compile the whole program and extract the LLVM bit code using `extract-bc`:

```
LLVM_COMPILER=clang make <target>
extract-bc <target>
```

2.3.3 KLEE Intrinsics in Build System Mode. KLEE provides intrinsics to write wrappers for self-contained UUTs. Those functions are declared in the KLEE header file but there are no definitions. When the test driver is compiled to LLVM bitcode directly through Clang, Clang generates function calls without attempting to link them. However, when compiled as a whole program, the linker complains that those functions are not implemented. Therefore, KLEE-Unit needs to supply stubs of those functions.

As mentioned in Section 2.2, we use cpp as the test file while parsing it as a c file. However, for the build system, they are treated as a C++ file. We find that Clang mangles the function signatures to the C++ style if the stub functions definitions are `static` (accessible only in the current source file), even if they are wrapped around `extern "C"`. KLEE cannot recognize the mangled names at runtime. Therefore, the function must not be marked as `static` . We also add the flag `__attribute__((noinline))` to the functions to make sure the function calls actually happen.

2.4 Run KLEE and Gather KLEE Test Cases

After obtaining the LLVM bitcode, KLEE-Unit invokes KLEE to perform the symbolic execution. We use the following arguments for KLEE:

```
--search=dfs                          \
--output-dir=<temporary directory>  \
--optimize                            \
--solver-backend=z3
```

The test cases are gathered asynchronously from the output directory and shown in the test case list at the bottom-right of the GUI (Figure 2) in real-time. The user may stop KLEE as soon as enough test cases are gathered. Therefore, the DFS search heuristic is used so that symbolic states can run to the end as soon as possible and generate corresponding test cases.

2.5 Generate Unit Test Cases

After the KLEE test cases are gathered into the GUI, the user can name some of them and translate them to unit test case written in testing harness. The translation is archived by rewriting the test driver AST using the rules in Table 1.

3 EVALUATION

In this section, we present two case studies of KLEE-Unit: generating test cases for a simple self-contained function `get_sign` in the single file mode, and for a more complicated real module, a least-recent-used (LRU) cache simulator module, in the CMake build system mode. Due to the limited time and resource, we do not perform a wide-range user study for KLEE-Unit, and the results presented here is somehow qualitative. However, they show the potential of how KLEE-Unit can help the user generate test cases with less manual effort.

3.1 get_sign

`get_sign` is a simple C function that returns 1, 0 or -1 based on the sign of the input number, shipped as a KLEE's example. The function has the following signature:

```
int get_sign(int x);
```

We use the single file mode in KLEE-Unit. The available templates for parameter `x` are `Symbolic` and `None`. We choose `Symbolic` for `x` and `Watch` for the return value. KLEE-Unit then generates the test driver in Figure 3.

```
void klee_unit_test_get_sign() {
  int x = SYMBOLIC(int);
  int ret = get_sign(x);
  WATCH(ret);
}
```

Figure 3: Generated test driver for `get_sign`.

We leave the driver as it is, and generate KLEE test cases using this driver. Figure 4 shows the translated KLEE driver, which is invisible to the user.

In one evaluation run, KLEE gives the three test cases in Table 3. We assigned the names shown in the table to these test cases.

Table 3: KLEE Test Cases and Assigned Names for `get_sign`.

	x	ret	Assigned Name
1	255	1	Positive
2	−2147483648	−1	Negative
3	0	0	Zero

```
#include <klee/klee.h>
#include "get_sign.c"

int main() {
  int x;
  klee_make_symbolic(&x, sizeof(int), "x");
  int ret = get_sign(x);
  klee_watch_obj(&ret, "ret");
}
```

Figure 4: Generated KLEE driver for `get_sign`. This piece of code is invisible to the user.

```
TEST_CASE("Positive") {
  int x = 255;
  int ret = get_sign(x);
  REQUIRE(ret == 1);
}

TEST_CASE("Negative") {
  int x = -2147483648;
  int ret = get_sign(x);
  REQUIRE(ret == -1);
}

TEST_CASE("Zero") {
  int x = 0;
  int ret = get_sign(x);
  REQUIRE(ret == 0);
}
```

Figure 5: Generated Catch2 test cases for `get_sign`.

And then, by clicking the translate button, KLEE-Unit generates the three test cases in the Catch2 harness shown in Figure 5.

In this whole process, the user does not need to write a single line of code. Certainly, a little effort is needed to identify the generated KLEE test cases and name them, but as discussed previously, we assume this effort is minimum if the user is using KLEE-Unit to practice TDD-like development strategies.

3.2 Least-Recent-Used Cache Simulator

`lru_cache` is a module simulating a cache with the least-recent-used replacement policy. It is built with CMake. A unit testing target (`lru_cache_test`) compiles the module, a mock main memory, and a unit testing file. We use KLEE-Unit to generate test cases for this module in the CMake build system mode. Here we show a more complicated test driver and the use of mock objects to show the flexibility of KLEE-Unit. The function we test is `cache_read_32` with the following signature:

```
int cache_read_32(Cache *cache, uint32_t address,
    uint32_t *value);
```

The cache object should be created by the `cache_create` function. If the cacheline exists in the cache, the function returns 0 waiting cycles. Otherwise, it brings up the cacheline from the mocked main memory and returns 50 waiting cycles. In either case, the data is written into the argument `value` which accepts the pointer of the variable and write to it.

The test code is in a different file from the production code, following the common practice of unit testing. We use code in Figure 6 as the starting point of the test file, which includes the necessary header and contains a helper function that resets the mock memory.

```
#include "lru_cache.h"
#include "cache_private.h"
#include "mock_mem.h"

void reset_mock_mem() {
    expected_read_count = 0;
    for (int i = 0; i < RECORD_COUNT; i++) {
        expected_read_ret[expected_read_count] =
     expected_read_count + 0x10000;
        expected_read_count++;
    }
    actual_read_count = 0;
}
```

Figure 6: Starting code of the test file for `cache_read_32`.

KLEE-Unit provides the following options for the test driver after analyzing the function signature shown in Table 4. We select the bold ones in the table. `cache` needs to be created by `cache_create` so we select `None` and fill it manually. `address` is input and `value` is output by passing pointers.

Table 4: Parameter Options for `cache_read_32`.

Type	Parameter	Option (Selected One in Bold)
`Cache *`	`cache`	Symbolic Array, Output Variable, In/Out Variable, **None**
`uint32_t`	`address`	**Symbolic**, None
`uint32_t *`	`value`	Symbolic Array, Output Variable, **In/Out Variable**, None

KLEE-Unit then generates the following driver template. To show the flexibility of the system, we make quite some changes to the driver, including creating the cache by calling `cache_create` manually, resetting the mocked memory, watching the actual access count, and duplicating the driver to make the call twice. These operations are very specific to the module we are testing. Figure 7 shows the test driver we get at last.

We create a 4-way 64-set LRU cache with 32-byte cacheline. We want to call the function twice to test both cache hit and cache miss. Lines 15-20 are basically a duplication of lines 6-12, which take a little effort to write. Line 9 and 17 are specific knowledge about the cache implementation.

It takes 9.7 seconds on average to compile the test driver and run KLEE on my machine (M1 Pro, adapted KLEE v2.3, release build, Z3 solver). Table 5 shows the two test cases generated by KLEE. We assigned the names shown in the table to these test cases.

These two test cases correspond to cache hit and miss respectively. Figure 8 shows the translated Catch2 test case of the hit case. The miss case is similar and omitted for brevity.

The generated test case sare directly written into the test file. We add the Catch2 header to the test file:

```
1  void klee_unit_test_cache_read_32()
2  {
3    reset_mock_mem();  /* added manually */
4
5    /* replacing Cache *cache = ? */
6    Cache *cache = cache_create(4, 64, 32, LRU);
7    uint32_t address = SYMBOLIC(uint32_t);
8    uint32_t value;
9    LET((address & 0x7FF) == 0);  /* added manually */
10   int ret = cache_read_32(cache, address, &value);
11   WATCH(ret);
12   WATCH(value);
13
14   /* duplicate the code piece above */
15   uint32_t address2 = SYMBOLIC(uint32_t);
16   uint32_t value2;
17   LET((address2 & 0x7FF) == 0);
18   int ret2 = cache_read_32(cache, address2, &value2);
19   WATCH(ret2);
20   WATCH(value2);
21
22   WATCH(actual_read_count);  /* added manually */
23 }
```

Figure 7: Final test driver to call `cache_read_32` twice. Lines 6-8 and 10-12 are generated, while the other are edited as explained in the comment.

```
TEST_CASE("Hit")
{
  reset_mock_mem();
  Cache *cache = cache_create(4, 64, 32, LRU);
  uint32_t address = 0x0;
  uint32_t value;
  CHECK((address & 0x7FF) == 0);
  int ret = cache_read_32(cache, address, &value);
  REQUIRE(ret == 0x32);
  REQUIRE(value == 0x10000);
  uint32_t address2 = 0x0;
  uint32_t value2;
  CHECK((address2 & 0x7FF) == 0);
  int ret2 = cache_read_32(cache, address2, &value2);
  REQUIRE(ret2 == 0x0);
  REQUIRE(value2 == 0x10000);
  REQUIRE(actual_read_count == 0x8);
}
```

Figure 8: Generated Catch2 test case for `cache_read_32` in the cache hit case. The cache miss case is similar and omitted for brevity.

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

And the test target can be built as it is:

```
> cmake
> make lru_cache_test
> ./lru_cache_test
===============================================
All tests passed (14 assertions in 2 test cases)
```

Table 5: Generated KLEE test cases and assigned names for `cache_read_32`.

address	ret	value	address2	ret2	value2	actual_read_count	Assigned Name
0xFEFF800	0x32	0x10000	0x1000000	0x32	0x10008	0x10	Miss
0x0	0x32	0x10000	0x0	0x0	0x10000	0x8	Hit

4 DISCUSSION

4.1 Source File Parsing

`pycparser` is not as robust as the parsers of gcc or clang. For example, it does not support C++-style comments. Very limited information is provided if the file has syntax errors. This is currently a limiting factor to the usability of KLEE-Unit. Also, as discussed in Session 2.2, KLEE-Unit allows a cpp file as the working test file to support modern unit testing harnesses, but can only parse it as a C file. Despite the lack of C++ support in KLEE (discussed in the next section), being able to parse C++ files will still help improve the usability of the system. For example, currently the system does not support existing Catch2 test cases to be in the test file. We write a dummy header trying to replace the Catch2 macros with empty stubs, but pycparser still fails to handle them.

4.2 Limitations of Symbolic Execution

The major problem that limits the stability of symbolic execution is path explosion—the number of states quickly explodes due to complex control flow. Arguably, symbolic execution is applicable for programs dominated by dataflow, such as arithmetic computations, where all possible values are computed "concurrently." On the other hand, diverging control flow causes problems since the symbolic state has to fork.

The path explosion problem also affects KLEE-Unit. Ideally, we want each unit test case to be representative for a class of equivalent input capturing the same semantics. However, this may not be possible due to the way how KLEE works. For example, when testing the LRU cache (Section 3.2), if line 9 and line 17 in Figure 7 are removed, KLEE will generate 64 test cases rather than 2. This is because the KLEE forks 32 states on array access into the cache sets. KLEE cannot determine which index to access and therefore forks on all possibilities. By enforcing the lowest 11 bits of `address` and `address2`, we are fixing the array index to 0 and therefore there is no state forking.

Such limitations of symbolic execution are so far the greatest limiting factor of the usability of KLEE-Unit. Even though the developer may have the insight about the implementation of the UUT, the effort needed to come up with such constraints may just go over the convenience that KLEE-Unit can bring. We do not expect such limitations before and it's one of the more important insights we gain from the project.

5 RELATED WORK

Applying symbolic (or concolic) execution to unit testing is not a new idea. Pex [9] (now now IntelliTest [7] in Microsoft Visual Studio) applies dynamic symbolic execution to C#. IntelliTest allows users to run symbolic execution on a selected C# method. It analyzes the method and generates input test cases, which users can view in a tool window and selectively save them as a test suite.

UC-KLEE [8] is another adaptation of KLEE that targets individual functions testing. Its name come from the fact that the input symbols are under-constrained, that is, an input may not be feasible when the function is integrated. To suppress false positive reports, UC-KLEE allows notations in the code for additional constraints. [5] tackles this challenge by extending test context to closely related functions. However, the false positive rates in these works are yet high. Also, these works do not address the test oracle problem but target bugs that can be automatically detected with checkers.

KLEE-Unit, from the implementation perspective, is similar to UC-KLEE. However, from the functionality perspective, it works more like IntelliTest as an test generator for programmers. The false positives are filtered by the programmers and test outputs are expected to be manually verified. Test cases are saved as concrete testing code, possibly integrated with common unit testing frameworks, so that they can be reused in the normal development process.

6 CONCLUSION

In this work, we present KLEE-Unit, a unit test generator driven by KLEE. The system works interactively with the user and provides flexibility in the generated test cases. Preliminary evaluation shows that the system helps reduce the effort to generate the same test cases manually written by the user. We discuss the limiting factors of the system's usability from the parser and the symbolic engine.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[2] The Qt Company. 2022. Qt 6 - The latest version of Qt. https://www.qt.io/product/qt6. Accessed: 2022-04-29.

[3] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2526–2540. https://doi.org/10.1145/3460120.3484813

[4] Martin Hořeňovský. 2022. A modern, C++-native, test framework for unit-tests. https://github.com/catchorg/Catch2. Accessed: 2022-04-29.

[5] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/3180155.3180253

[6] Zikai Liu. 2021. *Using Concolic Execution to Provide Automatic Feedback on LC-3 Programs*. Master's thesis. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/110284

[7] Microsoft. 2022. Overview of Microsoft IntelliTest. https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2022. Accessed: 2022-03-25.

[8] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[9] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *Tests and Proofs*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–153.