Universiteit Leiden

Master Computer Science

Automatically testing libraries symbolically

Name: Levi Vos
Student ID: s1668560

Date: 31-03-2022

Specialisation: Advanced Computing and Systems

1st supervisor: Dr. O. Gadyatskaya
2nd supervisor: Dr. K. Rietveld

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Automatically testing libraries is a critical task. Libraries offer programmers the net-benefit of being able to reuse existing code to solve difficult problems faster and with less bugs; in the off-chance the library contains a security vulnerability, however, this code-reuse feature potentially renders every program incorporating the library vulnerable as well. For this reason, it is of utmost importance to verify the correctness of libraries through testing. Automatic testing techniques allow library authors, as well as programmers incorporating the library in their own programs, to easily verify the library before shipping any code; potentially detecting and mitigating problems ahead of time.

Depsite its criticality, automatically testing libraries remains a challenging task as library functions can expect complex, pointer-rich datastructures as input. Moreover, it might be the case that there are internal dependencies among library functions, where one function sets some state required for another function to operate correctly. This thesis presents a novel method for addressing these challenges by making use of something we call the "batteries included"-assumption, which states that libraries provide all the prerequisites for valid use. Additionally, we evaluate the effectiveness of our method on a set of empirical experiments representing real-world code. This is done by evaluating our method on three libraries of differing sizes, of which two are taken from open-source code repositories. Based on the results of these experiments we discuss the strengths and limitations of our method, and compare our method with the related literature on state-of-the-art automatic testing techniques.

Contents

1 Introduction **2**
1.1 Thesis overview 3

2 Background **4**
2.1 Symbolic execution 4
2.2 Intermediate representation 5
2.3 Code coverage measures 6
2.4 Differences between automatically testing programs and libraries 7

3 Method **10**
3.1 Limitations of current methods 10
3.2 Method overview 13
3.3 Resolving state depencies 13
3.3.1 Function emulation 14
3.4 Generating test inputs through constraint solving 15
3.5 Differences with symbolic execution 16
3.6 Implementation 17

4 Evaluation **19**
4.1 Test subjects 19
4.2 Experimental setup 19
4.3 Evaluation results 20
4.3.1 Complexity evaluation 23

5 Discussion **25**

6 Related work **27**
6.1 Symbolic execution 27
6.2 Inferring library function preconditions 28
6.3 Method comparison 29

7 Conclusion **30**

Chapter 1

Introduction

Software engineering principles like "*don't reinvent the wheel*" [1] and "*standing on the shoulders of giants*" [2] propagate the use of libraries. These principles promote the use of libraries in order for a programmer to solve more complex tasks faster. According to these principles, failing to incorporate a library will result in *square wheels* with time and effort wasted. For this reason, it is considered good practice to reuse existing solutions as much as possible.

However, reusing code from a library is not without its own risks. If such a library contains a security vulnerability, this potentially renders all programs incorporating that library vulnerable as well. Moreover, the severity of the vulnerability gets multiplied for every program the library renders vulnerable. The recent vulnerability [3] in the popular Log4J logging framework is a catastrophic example of this happening; with computer security specialists stating it as one of the most critical vulnerabilities of the decade [4].

Despite the potential criticality of vulnerabilities in libraries, we are still convinced libraries provide a net-benefit to programmers; reinventing the wheel also leads to security risks, and its better to have one good and secure implementation, rather than many insecure ones. For this reason, we believe there to be a need for verifying the correctness of libraries by following the mantra of testing as much as possible.

Techniques for testing libraries have in the past mostly been limited to the test suites developed by library authors themselves. With the often repeated statements of how humans get tired and overlook potential problems, automatic testing techniques such as fuzzing and symbolic execution have been getting much attention. These techniques have proven themselves very capable of finding bugs in programs over the years [12, 14, 27, 36]. Unfortunately, due to differences between programs and libraries, these techniques do not automatically carry over to libraries as well.

One of the difficulties inhibiting automatic testing on libraries is that the interface of a library exposes a number of functions, which can be called in however many ways by a program incorporating it [20]. Contrast this with standalone programs which have only a single predefined entry point. To make

matters worse, it is often the case that there are internal dependencies among functions of the library, where one library function sets some state which is required for another library function to operate [7,20]. Testing a given library fully-automatically, therefore, also requires generating the test drivers which specify which library functions to call in what order with respect to these internal dependencies.

Hence the goal of this thesis is to find a way of automatically generating drivers for maximally covering a given library under test. This thesis contributes a novel method for automatically testing libraries through the use of constraint solving. Additionally, we show how automatic testing techniques can in fact benefit, rather than suffer from, internal dependencies among library functions.

1.1 Thesis overview

Chapter 2 contains background information for reading this thesis; Chapter 3 describes in detail how this research was conducted; Chapter 4 evaluates the results of the experiments; Chapter 5 discusses the strengths and limitations of our method; Chapter 6 gives an overview of related work; finally, Chapter 7 concludes this thesis.

Chapter 2

Background

This chapter contains background information for reading this thesis. In particular, we desribe symbolic execution; intermediate representations; code coverage measures; and, differences between testing programs and libraries.

2.1 Symbolic execution

Symbolic execution is a testing technique for analysing whether certain properties can be violated by a program [8]. Properties of interest include — among others — whether a division of zero is ever performed or whether memory may be referenced out-of-bounds. The goal of this technique is to execute as much paths through the program as possible, so as to say with as much certainty as possible whether these properties may be violated or not.

The way this is done is quite distinct from concretely executing the program. The key distinction made by symbolic execution is the ability to mark program inputs as *symbolic* — meaning that it can take on any value. This way, symbolic execution is able to execute all paths of the program which depend on the inputs being set to a particular value. Moreover, by forking the execution state on each conditional branch — constraining the values of the symbolic inputs in the resulting states such that the respective true or false branch is taken — multiple paths through the program can be executed concurrently.

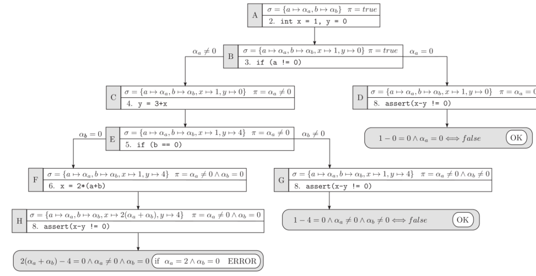
In contrast, a concrete execution can only execute a single path through the program based on its assigned input values. As such, symbolic execution is also quite distinct from other testing techniques, most notably fuzzing, which try to execute as much paths as possible by performing as much concrete executions as possible.

Symbolic execution is done by feeding the program through a *symbolic execution engine*. This symbolic execution engine takes the program — can be in some high-level language [35], some lower-level intermediate representation [12], or even lower-level bytecode [37] or binary [14, 36] format — and executes it by interpreting its statements as can be seen in Figure 2.1.

The symbolic execution engine maintains for each explored path: the *path conditions* which satisfy the taken branches along that path; and, a *symbolic memory store* which maps program variables to symbolic expressions or values [8, 15]. Encountered assignments update the symbolic memory store, whereas encountered conditional branches update the path conditions. By solving the path conditions — if these are satisfiable given the values in the symbolic memory store — using a *constraint solver*, it is possible to generate a concrete set of input values for traversing that path.

```
void foo(int a, int b) {
  int x = 1, y = 0;
  if (a != 0) {
    y = 3+x;
    if (b == 0)
      x = 2*(a + b);
  }
  assert(x-y != 0);
}
```

(a) Example function.



(b) Symbolic execution tree with each execution state labeled with an uppercase letter, the symbolic memory store σ, and the path constraints π.

Figure 2.1: Example function with corresponding symbolic execution tree (taken from [8]).

2.2 Intermediate representation

Ideally, we would like that modern testing tools are easily applicable to many different programs. However, these programs are likely written in different languages, since some languages are more popular than others, or some languages are more suited for the problem a program is trying to solve. Nonetheless, there are commonalities to be found among various programming languages. For example, many programming languages are compiled to the same runtime, be it native binary code or a virtual machine such as the JVM or BEAM. Moreover, compilation suites, such as GCC and LLVM, contain front-ends for a large number of programming languages. Here, one of the first steps these compilers perform is translating the high-level language toward an intermediate representation (IR). Therefore, all the front-ends implemented for the compilation suite, share the same IR during compilation such that they have a form more suitable for optimizations before being used to generate the object or machine code for a target machine.

For this reason, an IR is usually designed to be as generic as possible; typically, they tend to be free of specific high-level language features while also not making assumptions about features of the target machine. This generally characterizes itself in an IR of simple instructions, each representing one fun-

damental operation such as a load, store, addition, jump, etc... Figure 2.2 shows an example of code written in C with corresponding LLVM IR.

The use of such an IR is a common technique exploited by testing tools. The benefits of using an IR for testing are two-fold. First, IRs are designed to be shared among languages, therefore testing on this level will generalize to all languages which share the IR. Second, IRs are easier to analyse by design. The symbolic execution engine KLEE [12], for example, works at the level of LLVM bitcode, the IR of the LLVM compiler framework [26].

Another example can be seen with the symbolic execution engine, Angr [36], which is implemented on top of VEX, the IR of Valgrind [29]. Valgrind extracts VEX from binaries; since Angr also works on VEX, it is able to test any program written in any language as long as it compiles to binary. This way, enabling symbolic execution of programs without source code present, while at the same time not having to deal with difficulties regarding binary analysis.

Evidently, the decision at what level to implement the testing tool is important in how well the tool will generalize to other programs. If a tool is tailor-made for some high-level language it will not generalize to other high-level languages. However, this is not the only factor in deciding at which level to implement the testing tool. LLVM IR has mature tooling support and generalizes to a large number of programming languages. But on the flipside, it is obtained through the compiler and therefore requires the presence of source code to derive the LLVM IR, making it unable to test proprietary code. For this reason, projects like SecondWrite [6] lift LLVM IR from binaries, such that they can be symbolically executed using KLEE. However, due to the complexity of lifting binaries to LLVM this does not always result in a correct translation.

2.3 Code coverage measures

It is generally accepted that more rigorous testing is more likely to reveal bugs. The quality of a test suite can be objectively measured by the amount of code that has a test covering it. The test coverage survey by Hong Zhu et al. [44] describes various coverage criteria which have been proposed and studied for this purpose. In this section we briefly highlight a number of these criteria which are relevant to this thesis.

Consider the algorithm in Figure 2.2a and the corresponding Control Flow Graph (CFG) in Figure 2.2b as an example. The *line coverage* criterion is satisfied if all nodes of the CFG — and their corresponding instructions — have at least one test covering them. Full line coverage can be achieved in the example by calling `fib(0)` and `fib(2)`.

The *branch coverage* criterion is satisfied if all edges of the CFG have at least one test covering them. Branch coverage is stronger than line coverage because if all edges in the CFG have been covered, all nodes are neccessarily covered as well. Full branch coverage can in the example also be achieved by calling `fib(0)` and `fib(2)`.

Function coverage is one of the weakest criteria, as it is already satisfied if all functions have at least one test covering them. In the example this is already achieved by calling `fib(0)`.

The *path coverage* criterion is satisfied if all execution paths from the begin node to the end node of the CFG have at least one test covering them. Although path coverage is the strongest guarantee, it is too strong to be practically useful for most programs since there can be an infinite number of different paths in a program with loops. Full path coverage in the example is practically infeasible, as this would mean calling `fib` with every possible value an integer can take. On a system with 32-bit integers this would result in 2^{32} tests in order to achieve full path coverage.

Multiple condition coverage is a more pragmatic coverage measure, as it is weaker than path coverage but stronger than branch coverage. Like branch coverage, multiple condition coverage is satisfied if all edges of the CFG have a test covering them at least once. Additionally, however, all possible combinations which evaluate the condition to true or false, must have at least one test covering them. For example, consider the condition `if (x || y)`, multiple condition coverage is satisfied if there is at least one test covering the case where either `x` or `y` is true; both `x` and `y` are true; and, neither of them are true. In the example of Figure 2.2a, this criteria would be satisfied by calling `fib(0)`, `fib(1)` and `fib(2)`.

Note that within LLVM IR branch coverage and multiple condition coverage are effectively the same. During compilation to LLVM IR, conditions such as `if (x || y)` where multiple condition coverage must satisfy every possible combination of `x` and `y`, are transformed to independent branch conditions as shown in Figure 2.2. Here we count in the C code two conditional branches, namely: `if (n == 0 || n == 1)` and `for ( ; i <= n; )`. Whereas, the corresponding LLVM IR contains three conditional branches, namely at %1, %10 and %16. Hence by achieving full branch coverage on LLVM IR you get full multiple condition coverage for free as well.

2.4 Differences between automatically testing programs and libraries

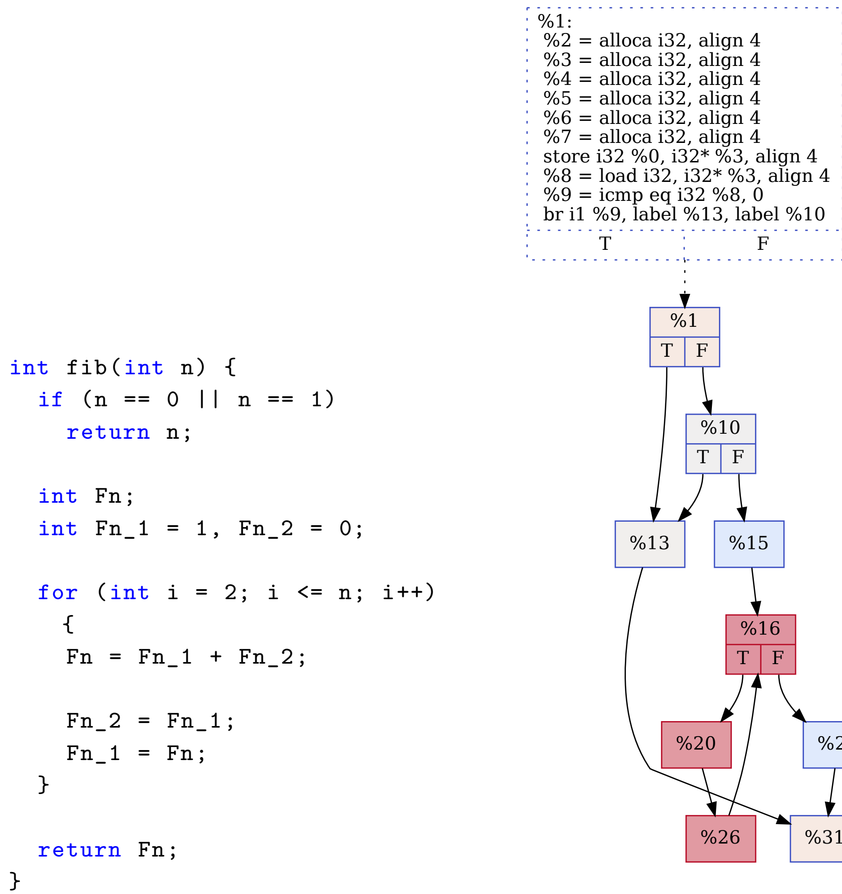
From a high-level point of view, programs and libraries are somewhat similar as they are both made up of executable instructions. When looking more closely, however, it becomes clear that they are different entities. Recent work has shown that automatic testing techniques have proven themselves very capable in finding bugs in programs [10–14, 27, 36]. However, because of the differences between the two, these same techniques do not easily carry over to automatically testing libraries as well. This section highlights some of these differences between programs and libraries so that it will be more clear how our method overcomes them in Chapter 3.

The largest difference is in the fact that all paths through a program have

a single predefined entry point, namely the main function. This main function accepts — by standard — as input arguments an integer, `argc`, and an array of strings, `argv`. As these inputs are of primitive types, it is clear what kind of values these inputs can take. Covering paths which are dependent on specific input values can then be done by generating these values either randomly or through techniques like constraint solving.

In constrast, a library does not have such a predefined entry point. Each of the functions a library exposes, is an entry point by themselves. Moreover, these library functions do not have standardized input arguments, but instead can expect complex, pointer-rich datastructures as inputs. One of the biggest challenges for automatically testing libraries, therefore, is in determining what kind of values to call the library functions with [18, 25, 32, 33, 35, 37, 38].

Additionally, it might be the case that the execution of a library function depends on state being set by some other function in the library. For this reason, automatic testing of libraries should have some strategy for determining these internal dependencies among library functions as well.




```
int fib(int n) {
  if (n == 0 || n == 1)
    return n;

  int Fn;
  int Fn_1 = 1, Fn_2 = 0;

  for (int i = 2; i <= n; i++)
   {
    Fn = Fn_1 + Fn_2;

    Fn_2 = Fn_1;
    Fn_1 = Fn;
  }

  return Fn;
}
```

(a) Bottom-up Fibonacci algorithm.

(b) CFG with instructions abstracted.

Figure 2.2: Bottom-up Fibonacci algorithm with corresponding CFG. For brevity, the LLVM IR instructions making up the nodes of the CFG have been abstracted as depicted by the dotted edge.

Chapter 3

Method

As mentioned in Chapter 1, the goal of this project is to automatically cover a given library under test. This chapter describes in detail our iterative approach to building the neccessary preconditions required for symbolically executing specific paths through the library.

3.1 Limitations of current methods

Before describing our method for testing libraries in detail, we would first like to provide a general note on testing libraries in the context of two previous approaches. This section highlights the methods of DART [18] and UC-KLEE [32, 33] in the context of specific library examples, to make a clear distinction where our work intends to contribute.

Both methods test individual library functions using constraint solving in order to maximally cover the paths through the function. Conditional branches provide a constraint on the path taken through the function. For example, when we encounter a conditional branch, such as an if-statement: `if (i < 5)`, for input `i < 5` the *then* branch will be taken, and for input `i >= 5` the *else* branch will be taken. Such a constraint on the control-flow within a program is known as a *path constraint*. Hence, by solving the path constraints of a program, we can direct test execution along untaken paths, leading to greater test coverage.

Where DART and UC-KLEE differ is in the following. Initially, DART performs a concrete execution with random inputs. During this random execution, the encountered path constraints are collected, and solved, to form the new inputs for the next concrete execution. Contrast this with UC-KLEE, which uses symbolic execution [9] to make an *interpretation* of the program with symbolic inputs, i.e. inputs can take on any value. UC-KLEE uses the popular symbolic execution engine, KLEE [12], to mark the function input as symbolic, and then solves the constraints in order to maximally cover the paths symbolically.

Now consider as an initial working example the linked list implementation in Figure 3.1. If we were to test this implementation using DART, then on the first concrete execution with `n == NULL` it encounters the constraint `n != NULL`,

```c
struct node {
  int val;
  struct node *next;
};
```

```c
int list_sum(struct node *n) {
  int sum = 0;

  while (n != NULL) {
    sum += n->val;
    n = n->next;
  }

  return sum;
}
```

Figure 3.1: Example linked list implementation (taken from [33]).

which is not taken since `n == NULL`. On the next execution, DART intends to cover the yet untaken branch. Therefore, solving the constraint `n != NULL`, will result in some random address other than `NULL`. Now a subsequent execution with the random address will probably crash, since it is most likely not a valid memory location. As a result, DART will mark this input as a crashing input. However, this is a false positive, since this library *implicitly* assumes only pointers to valid memory locations will be passed.

In contrast, UC-KLEE solves this problem using lazy initialization [22, 40]. Figure 3.2 shows the path exploration when applying symbolic execution with lazy initialization to the example linked list implementation. The gist of this technique is as follows. In order to explore paths with a path constraint containing a pointer, such as `n != NULL`, an instance of the pointee is allocated symbolically as if it was there all along. This process repeats infinitely with every iteration of the loop. In order to limit the number of allocations, k-bounding is used [16].

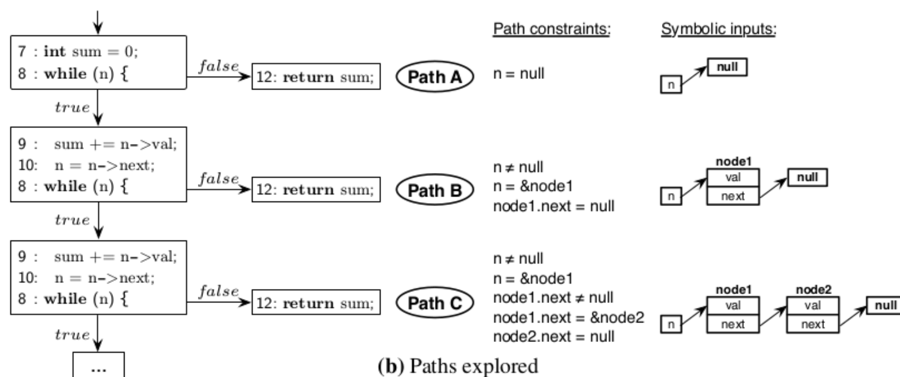


Figure 3.2: Example linked list implementation, analyzed by UC-KLEE using lazy initialization (taken from [33]).

UC-KLEE is able to generate complex, pointer-rich data structures using

lazy initialization. However, as addressed by the authors it may cause false positives. For example, consider the circular doubly linked list implementation in Figure 3.3. The defining property of a circular list is that the previous field of the first node refers to the last node; and in reverse, the next field of the last node refers to the first node. Without this property, the list would not be circular, and the function would not work correctly. Hence the assertions to check whether this property holds.

```
struct node {
  int val;
  struct node *next;
  struct node *prev;
};

struct dlist {
  struct node *bgn;
  struct node *end;
  size_t length;
};
```

```
int dlist_sum(struct dlist *dl) {
  int sum = 0;

  if (dl->length == 0) {
    return sum;
  }

  /* Two properties that must hold,
   * for a valid circular doubly
   * linked list.
   */
  assert(dl->bgn == dl->end->next);
  assert(dl->bgn->prev == dl->end);

  struct node *n = dl->bgn;

  do {
    sum += n->val;
    n = n->next;
  } while (n != dl->bgn);

  return sum;
}
```

Figure 3.3: Example circular doubly linked list implementation.

If we were now to test `dlist_sum`, assuming a lazily initialized `struct dlist *dl` pointer is passed, the first path is taken where `dl->length == 0`. The next path taken will be the one where `dl->length != 0`, which will result in an assertion failure, since UC-KLEE does not by default assign the circular property to the lazily initialized object. Therefore, this test result will be a false positive.

In all likelihood, since this test concerns a datastructure implementation, there will be a function present within the library such as `dlist_add_node`, which increments `dl->length` to a value `!= 0` while respecting the circular property of the list. Herein lies the crux of this thesis. When considering a single function there is little information about what properties should be respected within the library. Therefore, we propose to use as much as possible the functions exposed by the library itself, in order to solve the required path constraints. For example, a test for the constraint `dl->length != 0` would then

be satisfied through calling `dlist_add_node` before calling `dlist_sum`. We hypothesize, this approach leads to fewer false positives.

This approach is based on the following assumption about libraries. Namely, we assume all libraries are "batteries included" in the sense that they provide the prerequisites for valid use.

3.2 Method overview

In this section, we give a general overview of our method. The subsequent sections hereafter explain specific aspects of our method in more detail.

Previously we have described difficulties regarding automatically testing libraries. One of the difficulties is that the interface of a library exposes a number of functions, which can be called in however many ways by a program incorporating it. Moreover, it's often the case that there are internal dependencies among functions of the library, where one library functions sets some state which is required for another library function to operate. This state can be passed around the library functions through complex, pointer-rich datastructures residing in memory.

Key to understanding our method is our "batteries included"-assumption, which states that libraries provide all prerequisites for valid use. We assume that library authors abstract implementation-specific details behind these datastructures with good reason. If we were to tamper with these datastructures — for example by setting certain values within through the use of constraint solving — it is likely that this will result in a violation of implicit state contracts. Any bugs resulting from such a state violation will be deemed a false positive, as these are not possible through valid use.

Note that library functions are aware of these implicit state contracts, however. When there are internal dependencies among library functions, there is at least one library function which sets the state required for another library function to operate. By specifically exploiting these internal dependencies it is thus possible to bootstrap an initial state into another state, with which we can subsequently test the dependent parts of the library more thoroughly.

Therefore, we propose a bottom-up, dynamic programming approach instead. Our method starts out with a single valid state. Using copies of the initial state, we iteratively call the library functions to build up the subsequent states through function side-effects. With these subsequent states, we can then repeat this process. This way, we can continue bootstrapping states for as long as this results in new states.

3.3 Resolving state depencies

Given this general overview of our method, we propose the following bottom-up, dynamic programming approach for concretely resolving dependencies among library functions.

Consider a library as a set of functions, $\mathcal{F}$, and a set of global variables, $\mathcal{G}$. Each function has a list of parameters with a declared type. There are a number of paths which the function can take, depending on the input arguments in combination with the library state. We define state as the values in memory as well as the values of global variables at any moment of time. The values in these locations outlive a single function call and can thus influence subsequent function calls.

It is difficult to statically analyze what kind of values memory and global variables can hold at any moment of time, as this is inherently abstract. However, following our "batteries included"-assumption there should be functions in the library which set the required state. Therefore, we propose to emulate calls to library functions, such that we can keep careful track of side-effects to state.

3.3.1 Function emulation

Emulating library function calls is done as follows. Initially, for every parameter of every function in the library, we allocate a zero-initialized value in memory based on its declared type. Parameters with a pointer type are special in the sense that they can either point to NULL or to a valid underlying object. Therefore, in that case we allocate two pointer values; one pointing to NULL, and the other pointing to a valid underlying object. Together with the declared initial values of global variables this forms the initial state for emulation.

Now we emulate each function with every possible input argument in memory. Emulating a function is done by interpreting the statements making up the function with the input arguments from memory. This way, a function call is dynamically executed, through interpretation, so that we can record the side-effects to the state — if there are any. For every changed state we repeat this process (see Algorithm 1 for a pseudocode implementation of this algorithm).

Figure 3.4, shows this approach on a concrete example. Here we first emulate function `func` with `x = NULL`, which results in a crash. Therefore, we give up on this input argument. Next, we evaluate with `x = &object`, which succesfully covers a path through the function and changes the state with `object = 7`. Emulation with the newly changed state covers the second path through the function, but does not result in a new state; therefore, the algorithm is now finished.

We want to explicitly stress that our approach does not try to generate or solve the values making up the state. Instead, this approach zero-initializes the state in order to have a starting point from which we concretely transform the state only through emulating the instructions of the library functions. This is done such that our test cases do not violate any implicit state contracts, limiting the possibility of false positives.

Algorithm 1 Function emulation algorithm

```
1: newpath ← true
2: for i < iterations do
3:     for s ∈ states do
4:         for f ∈ F do
5:             args ← s.possibleArgs(f)
6:
7:             for a ∈ args do
8:                 copy ← State(s)
9:                 copy.addHistory(f, a)
10:
11:                                 ▷ Check if we have already evaluated this state.
12:                 if alreadyEvaluated(copy) then
13:                     continue
14:                 end if
15:                                 ▷ Emulate function call by executing LLVM IR.
16:                 emulateFunc(f, copy)
17:                 states.add(copy)
18:             end for
19:         end for
20:     end for
21: end for
```

3.4 Generating test inputs through constraint solving

Note that there are also functions which do not solely depend on state, but also on the supplied input arguments. As an example, consider the function in Figure 3.5 for calculating the n-th Fibonacci number. The path taken through the function depends on the supplied input argument n. Since this argument is of a primitive type — which is not stateful — we can easily generate inputs for the function using constraint solving. In our case, we use Z3 [5] to solve these kind of constraints.

Again we take a similar approach as described in Section 3.3. First, we emulate the function with a zero-initialized input argument: $n = 0$. However, since this input argument is of a primitive type, we additionally mark it as symbolic. During the emulation we encounter the conditional branch `if (n == 0)`, which we take since $n = 0$.

Now in order to cover other paths we query the constraint solver for a value $n \neq 0$, which possibly results in $n = 1$. Then we repeat the emulation for $n = 1$, encounter the conditional branch `if (n == 1)`, and generate a new argument by querying the solver for a value $n \neq 0, n \neq 1$.

Likewise, in order to cover the path where we perform one iteration of the `for`-loop, we query the solver for a value $n \neq 0, n \neq 1, i = 1, i < n$. Again we

```
// Current state of memory:
//   { x_1 = NULL,
//     x_2 = &object,
//     object = 0
//   }

//          x_1
//           _
//           |
//           |
//           |
//           v
void func(int *x) {
  if (*x == 7) {
    ;
  }
  else {
    *x = 7;
  }
}
//           _
//           |
//           |
//           v
//         CRASH!
// null pointer dereference
```

```
// Current state of memory:
//   { x_1 = NULL,
//     x_2 = &object,
//     object = 0
//   }

//          x_2
//           _
//           |
//           +<-----------+
//           |            |
//           v            |
void func(int *x) {  //   |
  if (*x == 7) {     //   |
    ;                //   |
  }                  //   |
  else {             //   |
    *x = 7;          //   |
  }                  //   |
}                    //   |
//                        |
//           |            |
//           |            |
//           +------------+
//        SUCCESS!
// repeat with changed state
```

Figure 3.4: Example how dependencies between two function calls can be resolved by keeping careful track of side-effects to state.

keep repeating this process for as long as there are new paths being covered.

Usually, path constraints are not solely dependent on state or symbolic input arguments; more often, it is some mixture of the two. When this is the case, we substitute the concrete values of the current state into the query passed to the solver. For example, when the stateful value `state->x` holds a concrete value — say 5 — and this value is constrained to be smaller than symbolic value s, then we solve by querying $5 < s$.

3.5 Differences with symbolic execution

At this point, there is a need for clarifying the difference between symbolic execution — as described in Section 2.1 — and our method. Both methods make use of interpretation for executing the program — or in our case library — statements, and both methods use constraint solving for generating test inputs. There are a few minor differences between the two methods, however. These differences are briefly highlighted in this section for completeness, but for the

```c
int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;

  int Fn;
  int Fn_1 = 1, Fn_2 = 0;

  for (int i = 1; i < n; i++) {
    Fn = Fn_1 + Fn_2;

    Fn_2 = Fn_1;
    Fn_1 = Fn;
  }

  return Fn;
}
```

Figure 3.5: Bottom-up Fibonacci algorithm.

remainder of this thesis we denote our method as being symbolic execution.

The most prominent differences between the two methods is in how conditional branches are handled. As described in Section 2.1, whenever symbolic execution encounters a branch conditional on some symbolic value, it forks the entire execution state — constraining the symbolic value such that the true or false branch is taken — and continues the interpretation of both paths concurrently. In contrast, our method does not interpret the paths concurrently, but instead does this sequentially. Our method emulates — or interprets with concrete input arguments from the state — the program statements. Whenever the input argument is of a primitive type, we additionally mark it as being "symbolic" — eventhough there is at the same time a concrete value for it — such that we can generate new values for it using constraint solving whenever we encounter the value in a branch condition. We continue emulating whatever branch is taken based on the concrete value. Since the value is also symbolic we, generate a value for it which satisfies the condition of the untaken branch and place this value in a copy of the state. We then visit the untaken branch with the modified state containing the solved value during a next iteration of our algorithm.

3.6 Implementation

This section describes the implementation of our method and the lessons learned in the process. Our goal was for our method to be applicable to as many libraries as possible. For this reason and to aid in ease of development — see Section 2.2 — we decided to implement our symbolic execution on top of an IR. Initially, we considered a number of different IRs for this. We chose the IR of

Egalito [39], a tool for comprehensive analyses and transformations directly on binaries, so that we would be able to test libraries without needing their source code present.

Things proved more difficult this way, however, and since our method is not fundamentally restricted to a specific IR, supporting a different IR would be a straightforward engineering effort. For this reason, we decided to implement our method on top of LLVM [26] IR instead. LLVM has mature tooling support and documentation which made it easier for us to get to a working proof-of-concept of our method.

As it stands, our method is implemented in a tool written in C++ of approximately 2000 lines of code. At its core, our tool works by emulating the LLVM IR instructions. In order to obtain the LLVM IR, each library was compiled using the clang compiler frontend. Our tool makes use of LLVM libraries in order to parse LLVM IR input files. Additionally, our tool makes use of the Z3 [5] constraint solver for solving path conditions.

Note that we have constrained our approach to the C programming language for now. Despite it being possible to generate LLVM IR from other languages — most notably C++ — this often results in more complex LLVM IR. Supporting these languages would again be a straightforward engineering effort not needed for a proof-of-concept. Given that C is the most popular language for writing libraries due to its easy interoperability with other programming languages, we believe this to be a reasonable compromise.

Chapter 4

Evaluation

In this chapter we evaluate the effectiveness of our method using a set of empirical experiments. Remember from Chapter 1 that we are mostly interested in maximally covering a given library. Hence by applying our method to three libraries, we aim to discover how well our method covers each library with regard to the code coverage metrics described in Section 2.3. Specifically, we are interested in the measured statement, branch and function coverage metrics of each library.

4.1 Test subjects

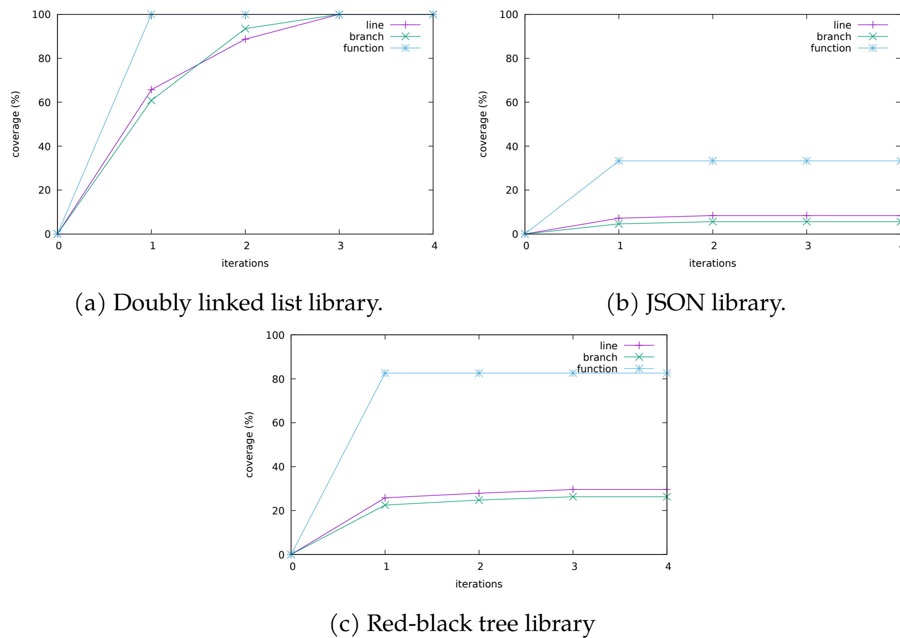
To evaluate the effectiveness of our method on real-world code, the libraries were selected to be of differing code sizes measured in lines of code (LoC). Moreover, two of the three libraries were taken from publicly available repositories on Github. The selected libraries consist of: a doubly linked list implementation, developed by ourselves, of approximately 150 LoC; a JSON parser and tokenizer [1] of approximately 400 LoC; and, a red-black tree datastructure implementation [2] of approximately 800 LoC.

4.2 Experimental setup

The experimental setup is as follows. Each library was compiled from source to LLVM IR through the clang compiler frontend. This LLVM IR forms the input to our tool together with an optional parameter how many iterations of our method — as described in Chapter 3 — to perform.

During symbolic execution our tool keeps track of which instructions, edges and functions are covered. By comparing against the total number available in the library: line, branch and function coverage can be determined, respectively. Recall from Section 2.3 that branch coverage on LLVM IR is in reality the same as

[1] `https://github.com/zserge/jsmn`
[2] `https://github.com/sfurman3/red-black-tree-c`



(a) Doubly linked list library.

(b) JSON library.

(c) Red-black tree library

Figure 4.1: Measured code coverage.

the stronger multiple condition coverage. For this reason, our measured branch coverage is in reality stronger than is normally the case in the literature. Additionally, our tool keeps track of the total number of states evaluated in the process. Taken together these statistics form the basis for the evaluation of our method.

4.3 Evaluation results

The results of this experiment can be seen in Figure 4.1. Here for each library the measured line, branch and function coverage is shown. As can be seen in Figure 4.1a, our method obtains full coverage on the doubly linked list library. However, comparing the figures in Figure 4.1 directly side-by-side, shows that there is a big discrepancy between the effectiveness of our method depending on the given library.

JSON library

The effectiveness — or lack thereof — of our method on the JSON library can be explained by looking at the function signatures of the JSON library API shown in Figure 4.2. The JSON library exposes two functions out of a total of six functions required for parsing and tokenizing JSON. The way this library is meant to be used is by first calling `jsmn_init` to initialize an object of type `jsmn_parser`, and then using that parser object in combination with a character string for

```
/**
 * Initialize JSON parser.
 */
void jsmn_init(jsmn_parser *parser);

/**
 * Run JSON parser. It parses a JSON data string into an array
 * of tokens, each describing a single JSON object.
 */
int jsmn_parse(jsmn_parser *parser,
               const char *js, const size_t len,
               jsmntok_t *tokens, const unsigned int num_tokens
    );
```

Figure 4.2: JSON libary exposed function signatures.

a call to `jsmn_parse`, which consequently fills an array of type `jsmntok_t` for every valid JSON token it encounters within the string. Thus far our "batteries included"-assumption as introduced in Section 3.1 holds nicely; as we don't know how to initialize objects of type `jsmn_parser` and `jsmntok_t`, this is handled correctly thanks to function calls to `jsmn_init` and `jsmn_parse`, respectively.

Where our method goes wrong, however, is in a slight ambiguity of the C programming language. Namely, when a pointer is passed as a parameter to a function there is no way of telling whether the pointer refers to a single object or an entire array of objects. Also note that a string in C is implemented as an array of characters. Our tool tries to make no assumptions regarding the inputs to functions. As a result, our method only calls the JSON parsing function with the string argument pointing to a single character. Self-evidently, parsing a single character does not cover a lot of code.

```
void idiomatic(int *array, size_t len) {
  ;
}
```

Figure 4.3: An example of idiomatic C code where the length of an array is placed directly after the array pointer parameter.

Mitigating this issue is an area where future work could improve upon. Observe that idiomatic C code, as can be seen in Figure 4.3, usually places the length of an array directly past the array pointer parameter. By using this as a heuristic it is possible to distinguish in some cases a pointer from an array.

To show this is indeed the case, we devised a little experiment. In this experiment, we manually create a test driver which our tool would also be able to generate automatically, with the only difference being that instead of a single

character we pass a character array to the `jsmn_parse` function. The test driver can be seen in the listing in Figure 4.4.

```
#include "jsmn/jsmn.h"
#include "klee/klee.h"

#define STR_SIZE 24

int main() {
  // These complex types are initialized through
  // the ``batteries included''-assumption.
  jsmn_parser parser;
  jsmntok_t tokens;

  char json_str[STR_SIZE];

  // Mark the character array symbolic.
  klee_make_symbolic(json_str, STR_SIZE, "json_str");

  jsmn_init(&parser);
  jsmn_parse(&parser, json_str, STR_SIZE, &tokens, 1);

  return 0;
}
```

Figure 4.4: Experimental JSON library test driver for use with KLEE symbolic execution engine to show the effect on coverage of passing a character array instead of a single character.

Next we evaluate this test driver using the KLEE symbolic execution engine and record the resulting line and branch coverage. To keep things mostly the same, we direct KLEE to use the Z3 solver for constraint solving and set a time limit of one minute. With only this small difference, KLEE — and thus also our tool if it would handle arrays — obtains 90.91% line coverage and 75.89% branch coverage.

Red-black tree library

The effectiveness of our method on the red-black tree library is also quite lacking compared to the exemplary results on the doubly linked list library — see Figure 4.1c and Figure 4.1a, respectively. Most strikingly, regardless of the number of iterations, four out of the twenty-three functions remain uncovered. These functions are only indirectly reachable through the exposed functions of the library interface. Taken together, these four functions are responsible for more than 33% branch and 50% line coverage, respectively. We hypothesise these functions — as well as the other uncovered parts of the library — are not being covered, because the required state is not built-up correctly.

To investigate this issue more thoroughly, we performed a manual code inspection of the library source code. Again, we found the "batteries included"-assumption to hold nicely; the red-black tree library exposes functions for valid initialization and modification of state. Where our method goes wrong, however, is in the following fundamental design limitation. Namely, our method expects state to be built-up in the memory and global data sections, as these sections are persistent inbetween function calls.

The red-black tree library does not adhere to this expectation, as the functions for modifying the state by, for example, adding or deleting a node, are characterized by the following signature:

```
new_root = RBT_state_modifier_function(root, ... )
```

We believe that the reason for this design choice is the self-balancing property of red-black trees, as rebalancing could result in a new root of the tree. Whenever our method calls a state modifying function on the tree pointed to by the `root` node object from memory, a `new_root` object is passed back to the stack section as a result. However, the stack section does not persist inbetween function calls in our method, meaning the carefully built-up state is then discarded. We did not foresee the passing of state like this when implementing our method. Unfortunately, this proves to be a shortcoming on this particular library.

4.3.1 Complexity evaluation

Now that we have shown that our method can obtain high coverage automatically — albeit with some modifications left for future work — we also want to evaluate whether our method scales to larger libraries. This section evaluates the complexity of our method in both time and space.

The pseudo-code implementation of our method as shown in Algorithm 1, shows that the time and space complexity is directly proportional to the number of states under evaluation. For every state, we evaluate all exposed functions of the library with all possible arguments, which in turn results in new states for further evaluation.

Figure 4.5 shows the number of states evaluated with every iteration of our method. Here we see that the number of states grows way faster with every iteration for larger libraries. In the worst case, such as with the doubly linked list library the number of states keeps growing exponentially. Extrapolating based on this observation, we can conclude that the exhaustive strategy for building state employed by our method does not scale to very large libraries.

However, this is a direction where future work can improve upon. For example, consider that very large libraries generally offer very much functionality; and, that with so much functionality, some parts of the library are possibly, less tightly-coupled with other parts of the library. For this reason, it might be possible to tame the complexity by identifying these parts of the library using

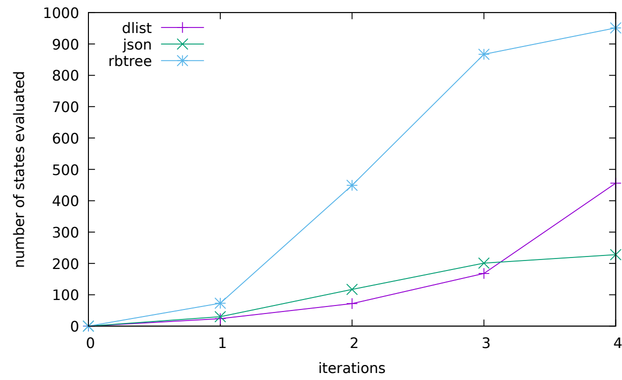


Figure 4.5: Number of states evaluated with each iteration of our method.

static analysis, and subsequently testing these decoupled parts exhaustively in isolation. Another possibility could be to analyse which parts are responsible for setting the state, such that these states can be built-up exhaustively before unleashing them on the remaining parts of the library.

Chapter 5

Discussion

This chapter discusses the strenghts and limitations of our method. The main contribution of this thesis is a method for testing libraries fully-automatically. We have shown how our method can benefit — instead of suffer — from internal dependencies among library functions. We specificly exploit these dependencies through the "batteries included"-assumption in order to initialize complex, pointer-rich datastructures. Moreover, Chapter 4 shows the validity of our method by evaluating its effectiveness on three libraries of different sizes.

This evaluation also exposes some limitations regarding our method. In Chapter 4 we observe a big discrepancy in the effectiveness of our method when comparing two libraries side-by-side. Here we note that our method is limited in disambiguating whether a pointer argument should point to a single object or to an entrire array of objects. This is problematic since arrays are frequently expected as input arguments for library functions.

This limitation can be addressed, however, through a simple heuristic. As can be seen in the code listing in Figure 4.3, it is often the case that whenever a function expects an array as input argument that there is an integer argument denoting the length of the array following the pointer argument. Future work can improve on our method by implementing this heuristic.

Another limitation is that our method does not make any assumptions on what values memory can hold. For this reason, our method does not use constraint solving in order to initialize the underlying objects referenced to by pointer arguments. This is a strength as we are convinced that solving complex structures will result in false-positives. However, this policy is too strict when the object under point is of a primitive type such as an integer or a character array. These primitive types can easily be set to specific values as dictated by the encountered path constraints. Future work can address this limitation by relaxing this policy a little bit.

As our method performs symbolic execution on LLVM IR there is a need for the source code to be available, since LLVM IR is obtained through compilation. Our method is further limited in that it only supports the code constructs encountered in C libraries for now. The need for source code could perhaps

be addressed by using tools such as SecondWrite [6] which is — although not perfect — capable of lifing LLVM IR from binaries. Another option would be to implement our method on another IR which is lifted from binary such as the ones from Valgrind [29] or Egalito [39]. In fact, this was how we first envisioned to implement our method, but this proved more difficult for creating a proof-of-concept.

Unlike other symbolic execution techniques (see Section 6.1), our method does not handle calls to the environment, such as system libraries or functions from other libraries than the target library, in any way. Our method is thus only able to automatically test libraries which are not dependent on any other libraries. Given that our method is mostly a proof-of-concept, we expect this to be a reasonable compromise.

The evaluation of our method is limited. Eventhough, we evaluated our method on three libraries of differing sizes, these libraries were all relatively small. Testing small libraries can still easily be done manually. Where automatic testing really shines, is in testing large and complex software, as it is the size and complexity which makes it more difficult to keep all edge cases in mind. Based on the performed experiments, it is unlikely that our method will scale to very large libraries. Given the fact that some widely-used libraries can easily be in the order of magnitude of hundreds of thousands lines of code, this shortcoming would be an interesting area for future work to improve on.

Chapter 6

Related work

This chapter places this thesis in the context of prior work done in the area of automatic software testing in general, and more specifically in the area of symbolic execution and automatic library test driver generation. Additionally, we discuss how our method relates to prior work and on what aspects our method differs.

6.1 Symbolic execution

Symbolic execution is a testing technique for systematically exploring many possible execution paths by abstractly representing inputs as symbols and resorting to constraint solvers to construct actual instances that would cause property violations [8]. Ever since its inception in the mid-1970s [9, 19, 23, 24], symbolic execution has become a widely popular testing technique, with Google Scholar reporting 742 articles that contain the exact phrase "symbolic execution" in the title as of August 2017 [8].

The survey on symbolic execution techniques by Baldoni et al. [8] provides an overview of the main ideas, challenges and solutions developed in this area. Some of these challenges faced in the area of symbolic execution can be found in how to handle symbolic data in memory; how to handle interactions with the environment when calls to library and system code can have side-effects; how to deal with the path explosion problem; and, the efficiency obstacle posed by solving non-linear constraints and interpretation. The highlighted techniques discussed hereafter deal with some of these challenges in novel ways or apply symbolic execution to novel domains.

When symbolic execution is used to analyse real-world applications, it often consumes all available memory in a relatively short amount of time due to the path explosion problem. Because of this memory pressure, KLEE prematurely terminates a substantial amount of paths as the given memory limit is reached. Memoization, by storing the early-terminated paths to disk and then replaying them later, is a solution for dealing with this problem. An extension [11] build on top of KLEE shows that this technique can enable KLEE to run large

applications for long periods of time. Sys [10] is another technique for dealing with large codebases. This technique works by first using static analysis for detecting potential error sites, before deeply examining these sites using symbolic execution.

Another impediment for using symbolic execution in practice is speed. Traditional symbolic execution engines such as QSYM [41], KLEE [12] and others, work by interpreting the program source code or some kind of IR. SymCC [31] obtains a speedup of two to three orders of magnitude over KLEE and QSYM, respectively, by compiling the symbolic execution right into the target binary.

The techniques described before require the presence of source code — or some IR obtained from the source code — for running symbolic execution. Angr [36] is a technique for running symbolic execution on binaries without the presence of source code. It does this by first lifting VEX, the IR of Valgrind [29], from the binary and running symbolic execution on top of this IR.

S^2E [14] is another technique for symbolically executing binaries by employing dynamic binary translation to directly interpret the x86 machine code. Additionally, S^2E is able to accurately deal with the environment by using virtualization to prevent lasting side-effects. This way different paths are not able to clobber each others domains.

The enormous popularity of smartphones also give rise to the need for testing mobile applications. Mirzaei et al. [28] show how to apply symbolic execution to test Android applications.

6.2 Inferring library function preconditions

The ability to reuse code from libraries is beneficial to programmers as it generally aids ease-of-development. Correctly using a library, however, can sometimes be challenging due to hidden dependencies between library functions which are not always clearly documented. The survey by Robillard et al. [34] describes various techniques for automatically inferring these kind of preconditions. Similarly, techniques for testing libraries — regardless of whether this is by fuzzing, symbolic execution or something else — also need to respect the preconditions for covering specific parts of the library. This section highlights a number of related works and the various methods they employ for dealing with this challenge.

The simplest method for dealing with this challenge is not to handle it at all. Techniques such as DART [18] and UC-KLEE [32,33] only test library functions in isolation. This way there is no need for inferring the required preconditions, but it comes at the cost of not being able to cover the deeper parts of a library which depend on the required preconditions.

In contrast, CUTE [35] acknowledges that there is a need for handling the preconditions when testing libraries. However, they require a human to specify the required preconditions of library functions, with the neccessary trade-off of giving up automaticity.

Other approaches, such as FUDGE [7] and FuzzGen [20] employ a corpus of code for mining patterns for correctly interfacing with the target library. This way they are only able to generate tests for parts of the library for which there is code available. Moreover, with FUDGE there is still a need for a human to tweak the inferred preconditions.

APICraft [42] is able to infer neccessary preconditions from closed-source libraries by employing both static and dynamic binary analysis and mining patterns from execution traces of programs incorporating the library.

Some other notable approaches use already existing test suites for deriving function preconditions [17, 21], or use natural language processing techniques for inferring these from the library documentation [43].

The technique most similar to our method, however, is the work done on RANDOOP [30] for feedback-directed random test generation. RANDOOP builds inputs by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. As soon as an input is build, it is executed. The result of this execution is the feedback which determines whether the constructed input satisfies the preconditions for that method call.

6.3 Method comparison

We want to highlight our method in light of the work done on DART [18], UC-KLEE [32, 33] and CUTE [35], as well as the challenges inhibiting automatic testing of libraries addressed in Section 2.4, in order to make a clear distinction where our method contributes.

All three of these works test library functions using constraint solving in order to maximally cover the paths through the function. DART and UC-KLEE are both able to automatically extract the interfaces of library functions and test them with the required inputs. However, DART only places constraints on integer types and cannot handle pointers and data structures. In this situation DART reduces to simple, random testing. In contrast, UC-KLEE improves on this regard by being able to generate complex, pointer-rich datastructures through a technique called lazy initialization [22, 40].

Both DART and UC-KLEE, however, test individual library function in isolation, without regard for internal dependencies among library functions. CUTE addresses this shortcoming, but requires the user to manually specify which library functions are related and what their preconditions are; hence, giving up on being fully automatic.

Our method complements these three methods by being able to automatically test library functions which depend on complex, pointer-rich datastructures, while at the same time respecting internal dependencies among library functions through a method similar to the work done on RANDOOP [30]. However, contrary to RANDOOP, our method adds constraint solving for solving specific values of branch conditions. Moreover, through our "batteries included"-assumption we specifically exploit these internal dependencies in order to initialize the values of these complex, pointer-rich datastructures.

Chapter 7

Conclusion

The goal of this thesis is to automatically cover a given library under test symbolically, while at the same time respecting the fact there might be internal depencies among library functions. These internal dependencies are specifically exploited in order to generate complex, pointer-rich input arguments through our "batteries included"-assumption. Furthermore, we have evaluated our method on a set of empirical experiments to determine the effectiveness of our method on real-world code.

Our method has shown to be viable on experiments with three libraries. These experiments exposed some limitations regarding our method which have been discussed and form the basis for future work to improve upon. Whether our method can be adapted to scale to very large libraries remains an open question for future work to expand on.

Bibliography

[1] `https://web.archive.org/web/20220103022036/https://en.wikipedia.org/wiki/Reinventing_the_wheel`,
last accessed on 2022-03-31.

[2] `https://web.archive.org/web/20220113061000/https://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants`,
last accessed on 2022-03-31.

[3] `https://web.archive.org/web/20220115054838/https://nvd.nist.gov/vuln/detail/CVE-2021-44228`,
last accessed on 2022-03-31.

[4] `https://web.archive.org/web/20220113133116/https://www.theguardian.com/technology/2021/dec/10/software-flaw-most-critical-vulnerability-log-4-shell`,
last accessed on 2022-03-31.

[5] `https://github.com/Z3Prover/z3`,
last accessed on 2022-03-31.

[6] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 295–308, New York, NY, USA, 2013. Association for Computing Machinery.

[7] D. Babić, S. Bucur, Y. Chen, F. Ivančić, 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*, pages 975–985, 2019.

[8] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, apr 1975.

[10] F. Brown, D. Stefan, and D. Engler. Sys: A Static/Symbolic tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216. USENIX Association, Aug. 2020.

[11] F. Busse, M. Nowack, and C. Cadar. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 63–74, New York, NY, USA, 2020. Association for Computing Machinery.

[12] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), dec 2008.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011.

[15] F. S. de Boer and M. Bonsangue. Symbolic execution formally explained. *Formal Aspects of Computing*, 33(4):617–636, 2021.

[16] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 157–166, 2006.

[17] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[19] W. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977.

[20] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[21] A. Kampmann and A. Zeller. Carving parameterized unit tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 248–249. IEEE, 2019.

[22] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] J. C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, apr 1975.

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[25] B. Korel. A dynamic approach of test data generation. In *Proceedings. Conference on Software Maintenance 1990*, pages 311–317. IEEE, 1990.

[26] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[27] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[28] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[29] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[30] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[31] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[32] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., Aug. 2015. USENIX Association.

[33] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 669–685, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[34] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2012.

[35] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[37] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, 2004.

[38] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 149–160, 2002.

[39] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, Jan. 2005.

[41] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[42] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu. APICraft: Fuzz driver generation for closed-source SDK libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2811–2828, 2021.

[43] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE, 2009.

[44] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, dec 1997.