UNIVERSITÄT SALZBURG

A Hybrid Symbolic Execution and Bounded Model Checking Engine in Selfie

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Science

by

Christian Edelmayer

Student ID 01617675

to the Department of Computer Sciences

at the Faculty of Natural Sciences

at the Paris Lodron University of Salzburg

Supervisor: Univ.-Prof. Dr.Ing. Christoph Kirsch

Salzburg, August, 2019

Statement of Authentication

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Salzburg, August, 2019

Christian Edelmayer

Abstract

In a world dependent on software, it is of the utmost importance to ensure the correctness of that software. For this reason, software needs to be analyzed. In this thesis, the software analysis methods called symbolic execution and bounded model checking are explained in detail. Furthermore, a hybrid symbolic execution and bounded model checking engine was implemented in Selfie. It is outlined thoroughly how the symbolic execution engine was extended into the hybrid engine. The main challenge was to implement the merging of paths which corresponds to bounded model checking. Our experiments show that symbolic execution produces formulas which can be solved more easily at the cost of a slower translation time. Exactly the opposite is true for bounded model checking. In other words, both methods have their advantages and disadvantages.

Contents

Chapter 1

Introduction

Nowadays, technology can be found essentially everywhere and (everyday) life without technology has become almost unimaginable. It can be discussed whether that is a good thing or a bad thing. However, it is a fact that humankind has become very dependent on technology and therefore also on software. For example, one just has to look at transportation: modern cars and airplanes contain a lot of software, which often consists of millions of lines of code [1]. Consequently, that means if the software in such systems is not correct, horrific things - like plane crashes - can and most certainly will happen [2]. Thus, it is of the utmost importance that the software we use is correct. "Correct" is a broad term, of course, and depends on the specific situation. Nevertheless, we could divide the possible states of a machine (on which the software/program is running) into good and bad states. Good states would represent the intended behaviour, i.e. the machine does what it should do correctly. Bad states would represent undesired behaviour in a specific situation, which may be a division by zero, illegal memory access, a wrong result, or any such issue. In essence, when a program is executed, every instruction of the program transitions the machine from one state to another. So, we could say, a correct program does not lead the machine into a bad state. This should make it reasonably clear what we want to avoid in software (transitions into bad states) [3].

In order to ensure the correctness of software, we need to analyze it. The software analysis methods called symbolic execution [4, 5] and bounded model checking [6, 7] were implemented in Selfie. As a matter of fact, a hybrid engine was implemented. The idea behind symbolic execution is to execute

a given program with symbolic (input) values in order to explore all possible paths, as opposed to just one path, of a program by using these symbolic values. Merging all of these paths corresponds to bounded model checking. Both these methods create formulas with which interesting properties - like the reachability of a division by zero - can be analyzed.

The main focus of this thesis lays on how the symbolic execution engine in Selfie was transformed into the hybrid engine. The first step was to unroll loops and recursion, i.e. to stop the relentless creation of new paths in such cases. Then, the actual merge of paths had to be implemented. Paths can only be merged at suitable locations, so a strategy was needed in order to determine that location. Also, the paths need to be executed in the right order and stopped at the right program location in order to be merged. What is more, all different components (registers, memory, etc.) of two paths have to be merged in such a way that no information is lost. Our engine is a hybrid since we can disable the merge, which corresponds to symbolic execution, and enable the merge (of all paths), which corresponds to bounded model checking.

Experiments conducted with this hybrid engine show that each method has its advantages and disadvantages. While symbolic execution produces formulas which can be solved more easily (faster), it is prone to the path explosion problem, which means that the creation of these formulas is slow if there are a lot of paths. On the other hand, bounded model checking counteracts the path explosion problem, but does increase the complexity of the formula, meaning it takes more time to solve that formula.

Chapter 2

Symbolic Execution & Bounded Model Checking

2.1 Motivation

Before introducing the mentioned methods, let us take a look at a very common approach for analyzing software/programs: simply executing the program. In general, that means executing the given program with some input and observing the behaviour of the program, i.e. checking whether some errors - for instance, a division by zero or the return of a wrong result - occur during the execution. With the explanation from the previous chapter in mind, that would correspond to a transition into a bad (machine) state. This approach should be quite familiar and well-known. It is a very simple approach and can sometimes prove to be effective [8]. However, most of the time the possible input for a given program is really big or even unbounded. Just imagine a program which checks whether a given number is a prime number. The possible input for this program would in theory consist of all natural numbers. Of course, there is always a practical limit, but the number of possible inputs would still be enormous. Therefore, it often is simply not feasible to test every possible input and exactly these "missed" (not tested) cases could cause errors. This quote from Edsger W. Dijkstra sums that up very nicely:

> "Program testing can be used to show the presence of bugs, but never to show their absence!" [9]

While being a relatively simple approach, which can be very effective sometimes, there is the downside of not being able to test every possible case. As usual, there is no free lunch, meaning that every method has its advantages and disadvantages and there is certainly not "the one" method which should be used in every case. However, taking the downsides of one of the most common approaches into account, the motivation to deal with different - and maybe even more formal - techniques and methods should be quite understandable.

2.2 Symbolic Execution

In order to understand the concept of symbolic execution, it is important to recall how the (concrete) execution of a program works. The machine model of a von Neumann architecture can be illustrated like this:

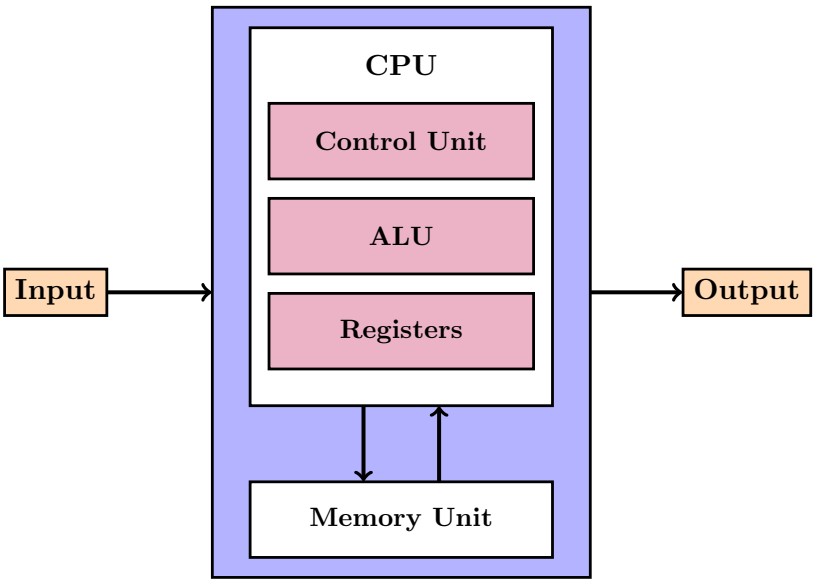

Figure 2.1: Machine Model

In essence, (concrete) input is given to the machine, the central processing unit (CPU) processes that input as specified by the program - or more precisely, by the instructions of the program - and produces an output. Both the program and the data are stored in memory. Each instruction has a very

small impact on the machine state and transitions the machine from one state to another. The state space of a machine is enormously large, nevertheless, everything is still completely deterministic [3].

Let us consider the following example:

```
int example() {
  ...
  x = 20;
  a = x * 2;
  if (a == 42) {
    return 1;
  } else {
    return 0;
  }
}
```

Figure 2.2: Simple Example

This code should be quite clear and understandable. The value `20` is assigned to the variable `x` and then the value of the variable `x` multiplied by `2` is assigned to the variable `a`. Afterwards, the `if`-statement checks whether the value of the variable `a` equals `42`. If that is the case, the value `1` is returned, otherwise the value `0` is returned. In the given case above, the execution is as follows: first, `20` multiplied by `2`, which equals `40`, gets assigned to the variable `a`. Afterwards, the condition of the `if`-statement is not `true` (since `40` does not equal `42`) and the value `0` is returned.

The interesting thing, which can be seen here, is that the execution can take different paths, and which path gets chosen depends on the value of the variable `a`, which in turn depends on the value of the variable `x`. In the `if`-statement, the given condition is evaluated and according to the result of this evaluation, the path - either the `if`-body or the `else`-body - is chosen. Hence, the execution follows exactly one path and not both. In principle, we

could see the `if`-statement as a point at which the execution has to take a choice: which path to follow.

It will be explained in more detail later how multiple paths emerge. To simplify it, normally `if`-statements and loops are the cause, since they translate to branch instructions which depend on a given condition that can either be `true` or `false`, however, that will be explained later. For now, it is sufficient to know that the execution of `if`-statements and loops depends on a given condition that determines which path the execution follows.

To come back to the mentioned problem with testing the program by just executing it with some input, it can easily be seen that whenever there is a choice between two paths, exactly one path is chosen. Of course, that is very reasonable, since each choice depends on a condition which can either be `true` or `false`. That is binary, a condition cannot be `true` and `false` at the same time. Thus, at each choice exactly one path is followed. If we remember all these choices and look at the complete path, which can consist of several subpaths, from start to finish, then we can also see that one complete path as essentially one path among many (complete) paths. However, normally there are a lot more (complete) paths than just one and that is the problem with software testing: when executing the program with some input, exactly one path is followed. It could be the case that only a few among many paths are followed, even if many different inputs are tested. One just has to think about a program where almost all of the input is handled equally except some corner cases. If the program is tested with "normal" input, then only a few paths are tested, while the other paths (corner cases) are not tested and can potentially cause errors.

This brings us to symbolic execution: the basic idea behind symbolic execution is to use symbolic values instead of concrete values [5]. Let us take a look at the simple example from before again, but this time with the value of the variable `x` being symbolic:

```
int example() {
  ...
  x = α;
  a = x * 2;
  if (a == 42) {
    return 1;
  } else {
    return 0;
  }
}
```

Figure 2.3: Simple Symbolic Example

In this case, α being assigned to the variable `x` simply means that the value of the variable `x` is symbolic as opposed to concrete, like in the example from before. The rest of the program is exactly the same as before: the value of the variable `x` multiplied by `2` gets assigned to the variable `a` and then the `if`-statement checks whether the value of the variable `a` equals `42`. If yes, the value `1` is returned, otherwise the value `0` is returned. However, since the value of the variable `x` is not concrete anymore, the execution differs. To put it very simply, if we say that a value is symbolic, it can be basically anything. For example, in our case the value of the variable `x` could be any number. Of course, within certain requirements. If the type is an integer, then it cannot be anything else than an integer, but it could be any integer. Anyway, these are formal details which are not important right now in order to understand the principle.

Now, the question arises how the (symbolic) execution of this example works, since the value of the variable `x` is symbolic. With the value of the variable `x` being symbolic, the value of the variable `a` becomes implicitly symbolic as

well. So, the condition of the `if`-statement depends on a symbolic value. It has to be checked whether the value of the variable `a` equals `42`. The value of the variable `a` is simply the value of the variable `x` multiplied by `2`. However, we have to keep in mind that the value of the variable `x` is symbolic, so that can be any number and is not further specified. If we resolve the value of the variable `a`, we can see that it represents the symbolic value multiplied by `2`, which essentially could also be just about any number. Consequently, we have to check whether a symbolic value multiplied by `2` equals `42`. Since we do not know what the symbolic value is, we cannot really make that decision. Let us reminisce that we could see this `if`-statement as a choice to make, namely which path to follow. And this is exactly what symbolic execution is about: since symbolic values are used, such decisions cannot be made in general. After all, how can we decide if an unspecified value multiplied by `2` equals `42`? The solution to this problem is very simple and interesting: we simply do not make such a decision.

Instead of taking the choice and following just one path, no decision is made and every path is explored. In our example, that would correspond to following the path of the `if`-body as well as following the path of the `else`-body. We could think of the execution and its paths as a tree structure, like this:

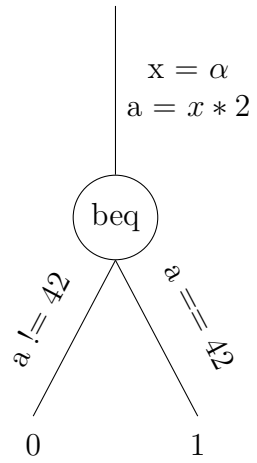

Figure 2.4: Execution Tree

In Figure 2.4 it can be seen that there are two paths which depend on the evaluation of the condition of the `if`-statement. Symbolic execution follows both paths when it encounters such decisions (`beq`-instructions with symbolic conditions). In order to follow not just one but both paths, a copy of the current program state at the instruction is made and adjusted accordingly. One path follows the `true` case of the condition and the other path follows the `false` case of the condition. Both paths need to remember their assumed conditions, also called constraints, i.e. which case of the condition they followed. For example, if a path follows the `else`-body, then the constraint would be that the condition of the `if`-statement is not true. The path condition is the combination of all these constraints on a given path. Let us take a look at the path that returns the value `1` at the end, as illustrated in Figure 2.4. Here the path condition would be $a == 42$. And resolving the value of the variable `a` would give us the following path condition: $\alpha * 2 == 42$. That is just a formula which can be satisfiable or not. In this case, it can easily be seen that it is satisfiable with $\alpha == 21$. If we look at the other path, the path condition would be $\alpha * 2 \; != 42$.

The path condition is simply a formula which (in general) depends on the symbolic values and is either satisfiable or not. The program can by analyzed with the path condition. For example, if a path encounters a division by zero, the path condition at this division by zero can be used in order to check whether the division by zero is reachable. If the path condition is satisfiable, then the path condition, which is just a formula, can be solved and concrete values can be calculated which lead the execution to this division by zero.

Another example would be to analyze the result of a program. After a path has finished its execution, the path condition could be combined with some additional constraint, for instance, that the return value equals `42`. The conjunction of the path condition and the additional constraint is also a formula. If this formula is satisfiable, then we know that there is a case where the program returns the value `42`. What is more, we do not only know that this case exists, solving the formula also gives us the concrete input for that case.

Of course, symbolic execution is not limited to these two examples. There are a lot more interesting properties which can be analyzed. In reality, path conditions are not as simple as illustrated in the example before, therefore external software exists which can solve these formulas for us. However, more information about solving the path conditions and about path conditions in general will be given in the next chapter. As of now, it is important to understand just the basic principle.

When looking at the machine model again, but this time during symbolic execution, it looks as follows:

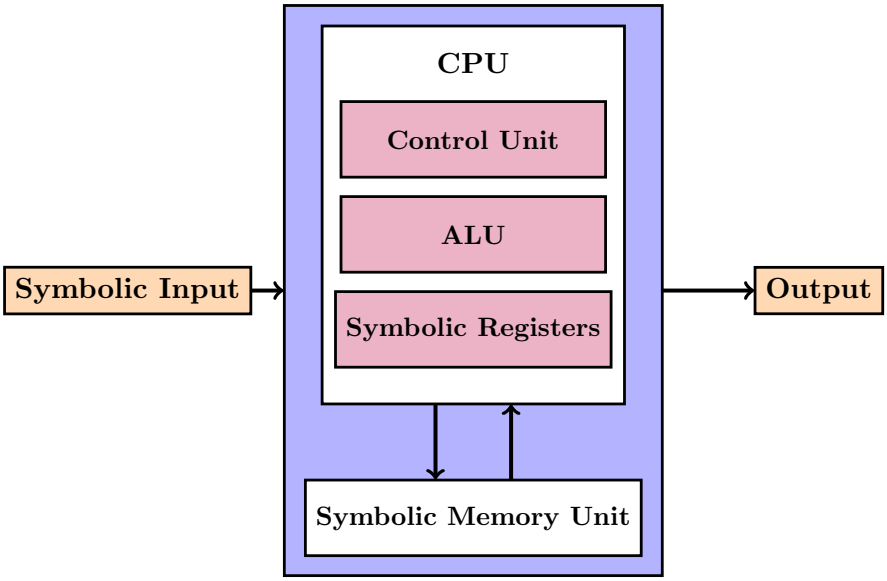

Figure 2.5: Symbolic Machine Model

The input is symbolic and that implicitly makes the memory as well as the registers symbolic, since they both depend on the input. In essence, different paths represent different program states. As outlined before, if a new path is encountered, then the program state is copied and adjusted, so that for each path the program state and therefore also the machine state is defined.

To sum up, the concept of symbolic execution means that symbolic (input) values are used instead of concrete values, the program is executed with these symbolic values by exploring all paths and each path has its own path

condition, depending on the symbolic values and the assumed conditions at branch instructions. These path conditions can either be satisfiable or not. If a path condition is satisfiable, then concrete values can be calculated.

2.3 Bounded Model Checking

In general, programs are quite complex and extensive, meaning that they consist of a lot of `if`-statements and loops, which increase the number of possible paths. This is a fundamental problem of symbolic execution and also known as "path explosion" [10], since the goal of symbolic execution is to explore all paths.

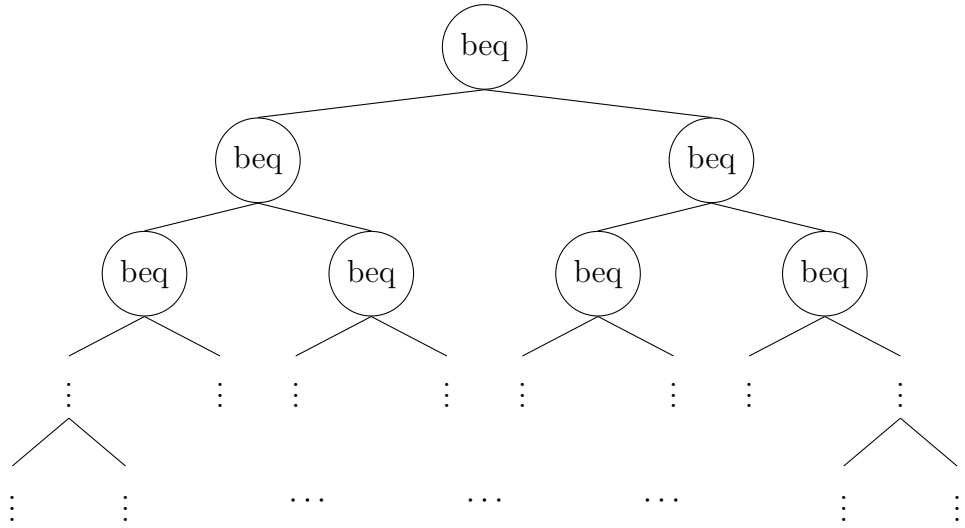

Figure 2.6: Path Explosion Problem

Figure 2.6 illustrates this problem. A `beq`-node simply represents a point from which two possible (execution) paths emerge. Usually, the number of paths grows exponentially with the size of the program [10]. The more paths there are to explore, the longer the symbolic execution takes.

As programs tend to get quite large and complex, this problem needs to be addressed properly. To begin with, let us take a look at a simple example again:

```
int example() {
  ...
  x = α;
  a = x * 2;
  if (a == 42) {
    a = 2;
  } else {
    a = 1;
  }
  a = a - 1;

  return a;
}
```

Figure 2.7: Simple Merge Example

This example uses symbolic values again, but is now a bit different than before. We can see that no matter whether the path of the `if`-body or the path of the `else`-body is taken, the program code after the `if-else`-statement is the same. As explained before, symbolic execution follows both the path of the `if`-body and the path of the `else`-body, which means that the code for decreasing the value of the variable `a` by `1` and returning the new value of the variable `a` would be executed in both paths, even though it is exactly the same for both of them. It would be better if this code after the `if-else`-statement was executed just once, since it is exactly the same for both paths. The value of the variable `a` is different, but the instructions are the same. One idea would be to somehow merge the paths together after the `if-else`-statement, as from this point onwards their execution does not differ. Only the value of the variable `a` differs. If we assume that we can somehow merge the paths together after the `if-else`-statement, the execution tree would look like this:

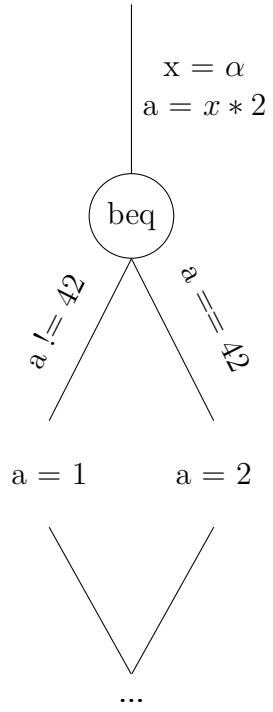

Figure 2.8: Bounded Model Checking

Merging paths at suitable locations would reduce the number of paths and counteract the path explosion problem. That is the idea behind bounded model checking. Merging all paths corresponds to bounded model checking [10]. In other words, over the course of the (symbolic) execution all created paths are merged together again, so that at the end only one path remains, which represents all paths merged together. That greatly reduces the number of paths, but does increase the complexity of some other aspects. How the merging exactly works and what the increase of complexity means, will be outlined in detail in the next chapter. The important thing to understand at this point is, that bounded model checking corresponds to merging all paths during symbolic execution.

Chapter 3

Implementation

3.1 Selfie

3.1.1 Introduction

The (hybrid) engine is implemented in Selfie[1]. Selfie is a project with the main purpose of teaching computer science and was developed at the Department of Computer Sciences of the University of Salzburg in Austria by the Computational Systems Group[2]. It is written in C* which is a small subset of the programming language C [11], and it is completely self-contained. Selfie consists of about ten thousand lines of code (as of: August 2019) and is implemented in a single file (`selfie.c`). A lot of emphasis is placed on identifying self-referentiality. While Selfie is being kept intentionally minimal in general, all its principles are absolutely realistic. The implementation includes

- a self-compiling compiler
- a self-executing emulator
- a self-hosting hypervisor
- a symbolic execution engine

and much more [12] [13].

[1] `https://github.com/cksystemsteaching/selfie`
[2] `http://www.cs.uni-salzburg.at/~ck/`

3.1.2 Monster

In this section the symbolic execution engine called "monster", which was already implemented in Selfie, is explained. First of all, it is not necessary to know all the implementation details of Selfie in order to understand monster and the hybrid engine. However, some things are important to understand. To begin with, we need to understand what monster is.

Monster is basically an emulator implementing a subset of the instruction set architecture RISC-V[3], called RISC-U. It can execute RISC-U code by interpreting it. In other words, monster emulates a RISC-U processor in software. The emulated machine is a von Neumann machine, as it was illustrated in chapter 2. Monster creates an instance of that machine in software. Also, monster works with contexts. A context is a concept in Selfie which we could see as a representation of (a part of) the machine state. Each context has (in principle) its own registers, its own memory space and its own program counter. If we want to execute a program with monster, we first need a context. Therefore, a context is created and initialized. Then, the program (binary) is loaded into the context. Now, this context needs to be executed by monster. In order to do that, a context switch is performed, which loads the context into monster. Afterwards, monster can execute the context, which means executing the code of the program stored in the context. The execution follows a fetch-decode-execute cycle. An instruction is fetched, decoded and executed (interpreted) [14].

We also need to know that C* supports only five different statements [13]:

- assignment
- while
- if
- procedure call
- return

[3] `https://riscv.org/`

Only the `if`-statement and the `while`-statement translate to `beq`-instructions in RISC-U, that means only these two statements can create new paths. The symbolic execution engine is able to execute RISC-U code symbolically up to a given number of instructions. This number can also be specified as unbounded. Only the `read` system call gives symbolic values. In other words, if the symbolic execution engine encounters a `read` system call, then it assumes that the read value is symbolic. For example, user input could be requested with a `read` system call. When symbolically executing that code, the user input would be assumed to be symbolic, meaning it is unspecified and each read byte can be anything from 0 up to 255 [12].

Some components of the context are a bit different when it is executed symbolically, like the memory store. However, this will be explained later and is not that important at this moment. The only thing we need to understand is that each context has its own path condition, which is written in the SMT-LIB language [15] and stored as a string. The SMT-LIB language is simply a specification on how to write certain things, like the path condition. It is not that relevant how the path condition is written, we just need to know what we have learned about the path condition in chapter 2. Now, let us take a look at how the symbolic execution of a program with monster works exactly.

First, a new context is created and initialized. The code of the program, which was compiled by the selfie compiler, is loaded into the context and the context is loaded into monster. Then, the instructions of the program are executed. If a `read` system call is encountered, the read value is assumed to be symbolic. For example, if user input is requested by a `read` system call and stored in a variable, then the value of that variable would be symbolic. Symbolic values are represented and stored as strings (written according to the SMT-LIB language). That means, if a symbolic value is stored in a register or in the memory, then simply a string representing that symbolic value is stored. Each new constraint on a symbolic value is also represented as a string, mostly by modifying and extending the existing string. An instruction with symbolic values may cause new constraints on the symbolic values, for example, if a symbolic value is used in an addition, that addition would be a constraint on the symbolic value and therefore encoded in the string. Also, it is possible for a concrete value to become symbolic. For instance, if a concrete value is multiplied by a symbolic value, then the concrete value becomes implicitly symbolic and is also represented as a string from that

point onward. Here we can see why the memory space of a symbolic context needs to be modified a bit.

If the execution encounters a `beq`-instruction whose condition depends on symbolic values, things get interesting. This instruction represents a point from which two paths emerge. One path for the `true` case of the condition and one path for the `false` case of the condition. Since the condition depends on symbolic values, it cannot be decided whether the condition is `true` or `false`. Here is what happens: monster creates a copy of the existing context. One context follows the path of the `true` case of the condition and the other context follows the path of the `false` case of the condition. Of course, monster adjusts the path condition and the program counter of both contexts accordingly.

Basically, that is how the unextended monster engine works. After a context has finished its execution, it writes its path condition into a specified file (written in SMT-LIB language). In our setup, this path condition would then be given to an external software called Boolector [16] which is a solver for such formulas and generates concrete solutions (inputs), if the formula is satisfiable.

Important to understand is that for each path the whole machine state - represented by the corresponding context - exists.

Whenever a new path is encountered, a new context is created in order to follow that path. We could say that each context represents a path of the execution tree and when we talk about merging paths, we are essentially talking about merging contexts.

3.2 Algorithm

The main goal was to extend the existing symbolic execution engine (monster) in Selfie by implementing the following algorithm:

Input: Choice function *pickNext*, similarity relation $\sim$, branch checker *follow*, and initial location l_0.
Data: Worklist w and set of successor states S.
$w := \{(l_0, true, \lambda v.v)\}$;
while $w \neq \emptyset$ **do**
  $(l, pc, s) := pickNext(w); S := \emptyset$;
  // Symbolically execute the next instruction
  switch $instr(l)$ **do**
    // assignment
    case $v := e$ **do**
      $S := \{(succ(l), pc, s[v \rightarrow eval(s, e)])\}$;
    // conditional jump
    case *if(e) goto* l' **do**
      if *follow*$(pc \wedge s \wedge e)$ **then**
        S := $\{(l', pc \wedge e, s)\}$;
      if *follow*$(pc \wedge s \wedge \neg e)$ **then**
        S := $S \cup \{(succ(l), pc \wedge \neg e, s)\}$;
    // assertion
    case *assert(e)* **do**
      if *isSatisfiable*$(pc \wedge s \wedge \neg e)$ **then**
        abort;
      else
        S:= $\{(succ(l), pc, s)\}$;
    // program halt
    case *halt* **do**
      print *pc*;
  // Merge new states with matching ones in w
  foreach $(l'', pc', s') \in S$ **do**
    if $\exists (l'', pc'', s'') \in w : (l'', pc'', s'') \sim (l'', pc', s')$ **then**
      $w := w \setminus \{(l'', pc'', s'')\}$;
      $w := w \cup \{(l'', pc' \vee pc'', \lambda v.ite(pc', s'[v], s''[v]))\}$;
    else
      $w := w \cup \{(l'', pc', s')\}$;
print "no errors";

Algorithm 1: Generic Symbolic Exploration [10]

A detailed description of this algorithm can be found in [10]. Basically, it is a generic algorithm for the symbolic exploration of programs. In simplified terms, it assumes that a function for picking the next state, a similarity relation for checking whether states can be merged, and a branch checker for deciding whether to follow a branch are already defined. A state consists of the program location l, the path condition pc and the symbolic store s. The symbolic store is simply a mapping from variables to corresponding values, which can either be concrete or symbolic. The worklist of states is worked through until there are no more states left. The algorithm distinguishes different kinds of instructions: assignments, conditional jumps, assertions and program halts. Especially interesting is the case of the conditional jump: the branch checker *follow* decides whether to follow a branch or not. Also, in the last few lines of the algorithm it is checked whether a state can be merged with another state. If two states are at the same program location, the similarity relation $\sim$ is needed in order to determine whether a merge is possible [10].

A state of the algorithm corresponds in our case to a context. The memory of the (symbolic) context is the symbolic store. The program location of the context is given through the program counter and each context stores its path condition as a string. Furthermore, the branch checker *follow* can be seen as the decision whether to create a new context at a branch instruction.

In order to implement this algorithm, the assumed input (function for picking the next state, similarity relation for checking whether states can be merged, branch checker for deciding whether to follow a branch) needed to be implemented as well. Particularly, the existing symbolic execution engine in Selfie was extended by the following features:

- unrolling loops/recursion
- merging paths
- detecting divisions by zero
- detecting invalid memory access

All of these features and how each implementation works will be outlined over the course of the next few sections.

3.3 Unrolling Loops/Recursion

While the existing symbolic execution engine already implemented a lot of functionality, it did not work with loops and recursion, if the condition depended on symbolic values and the maximal execution depth was defined as unbounded. In order to understand this, we have to take a look at how loops translate to `beq`-statements in RISC-U code:

```
// example snippet
int example() {
  ...
  while (a < 42) {
    a = a + 1;
  }
  // something
  ...
}
```

```
ld t0,-8(s0)
addi t1,zero,42
sltu t0,t0,t1
beq t0,zero,6    // check condition
ld t0,-8(s0)     // loop body
addi t1,zero,1   // loop body
add t0,t0,t1     // loop body
sd t0,-8(s0)     // loop body
jal zero,-8      // jump back to
                 // loop condition
ld t0,-8(s0)     // outside of loop body
addi t1,zero,1
sub t0,t0,t1
```

Figure 3.1: RISC-U Translation

We can see, in principle, that the `beq`-instruction checks the condition of the loop. If the `beq`-instruction depends on symbolic values, a new context is created and one context branches, the other one does not, so ultimately both paths are followed. However, the context which follows the path inside the loop body encounters a jump back at the end of the loop body, and executes the `beq`-statement, which checks the loop condition, again. At this point, a new context is created again, since the condition still depends on symbolic values. This process repeats again and again, which means that the symbolic execution engine does not terminate. Recursion is a bit more complex, but the principle is almost the same as with loops and therefore the engine does not stop execution either. That was exactly the problem with loops and recursion in the original version of monster.

The solution for this problem was to introduce some criterions to stop the relentless creation of new contexts in such cases. For this purpose, two criterions were used:

- maximal execution depth
- limit of `beq`-instructions

The maximal execution depth was already implemented in the original version of monster, but needed to be adjusted accordingly. The maximal execution depth defines the maximum number of allowed instructions per context. Each context stores its own execution depth and when a new context is created (copied from another context), then the execution depth is copied as well. So, for each context the current execution depth is known and increased by every executed instruction. If a context exceeds the maximal execution depth, it is terminated by monster, which makes sure that each context executes no more instructions than permitted. The adjustment of the maximal execution depth was the first step to unrolling loops and recursion.

However, it is not that easy to decide a reasonable maximal execution depth, since often many instructions are needed to properly explore the program, but too many instructions may cause an enormous path explosion. Therefore, another limit had to be implemented.

A new context is only created at a `beq`-instruction with a symbolic condition. The idea is that each context stores a counter which is incremented when the context creates a new context at a `beq`-instruction. If this counter of the context has reached a given limit and the context encounters a `beq`-instruction with a symbolic condition, monster makes sure that it does not create a new context anymore and that it just follows the path of the `true` case.

With the implementation of these two limits, the symbolic execution engine is able to handle loops and recursion quite well now. Also, the implementation of the limit on `beq`-instructions corresponds to the follow decision from the algorithm before.

3.4 Merging Paths

Implementing the merging of paths was the most difficult feature to implement, since there is a lot to be considered. Firstly, only contexts at the same program location can be merged. For example, it would not make any sense to merge a context which is currently in an `if`-body with a context which is currently in an `else`-body. Secondly, this program location needs to be known beforehand, in order to stop the contexts from executing and "missing" the merge location. So, when a context is created, the right merge location has to be found and the context has to be stopped from executing when it has reached that location. And on top of that, the actual merge has to be implemented.

3.4.1 Finding the Merge Location

If a `beq`-instruction with a symbolic condition is encountered and the before mentioned limits have not been reached, then a new context is created in order to explore both possible paths: the `true` case and the `false` case of the `beq`-instruction. As outlined before, in our particular setup in Selfie that would represent an `if`-statement or a `while`-loop. In other words, one context assumes that the condition is `true` and therefore executes the body of the `if`/`while`-loop and the other context assumes that the condition is `false` and therefore does not execute the body of the `if`/`while`-loop (but the body of the `else`, if there is one). While the contexts are in different program locations, they cannot be merged. However, as soon as they are at a same program location again, they can be merged. That would be after the `if`/`else`-body or after the loop body, respectively.

In order to stop the context from executing at the right program location, the location at which a merge is possible has to be known. Where a merge is possible, depends on whether the `beq`-instruction represents an `if`-statement, an `if-else`-statement, or a `while`-loop.

So, at first we need to determine what the `beq`-instruction represents. The following strategy is used:

Strategy: Examine last instruction before target location of the `beq`-instruction

- no `jal`-instruction: end of `if` without `else`-branch
- `jal`-instruction with positive immediate value: end of `if` with `else`-branch
- `jal`-instruction with negative immediate value: end of `while`-loop body

If the last instruction before the target location of the `beq`-instruction is not a `jal`-instruction, then it represents an `if`-statement without an `else`-branch. In that case, the merge is possible after the `if`-body. If the last instruction before the target location of the `beq`-instruction is a `jal`-instruction with a positive immediate value, then it represents an `if`-statement with an `else`-branch. In that case, the merge is possible after the `else`-body. If the last instruction before the target location of the `beq`-instruction is a `jal`-instruction with a negative immediate value, then it represents a `while`-loop. In that case, the merge is possible after the body of the loop.

Let us take a look at how `if-else`-statements and `while`-loops translate to `beq`-instructions in Selfie in order to understand this strategy.

```
// example snippet
int example() {
  ...
  if (a == 42) {
    a = 2;
  } else {
    a = 1;
  }
  // something
  ...
}
```

```
ld t0,-8(s0)
addi t1,zero,42
sub t0,t1,t0
addi t1,zero,1
sltu t0,t0,t1
beq t0,zero,4    // check condition
addi t0,zero,2   // if body
sd t0,-8(s0)     // if body
jal zero,3       // jump to skip else
addi t0,zero,1   // else body
sd t0,-8(s0)     // else body
ld t0,-8(s0)     // outside of if/else
addi t1,zero,1
sub t0,t0,t1
```

Figure 3.2: RISC-U Translation

The `beq`-instruction checks whether the value of the register `t0` equals the value of the register `zero`, which is always zero. If so, then the execution branches to the target location, which would mean a jump of `4` instructions forward, as the third value of the `beq`-instruction indicates. Otherwise, the next instruction is executed. In essence, the `beq`-instruction represents the check of the condition of the `if-else`-statement. The last instruction before the target location of the `beq`-instruction is a `jal`-instruction with a positive immediate value, which represents the jump to the end of the `else`-branch. That makes sense, because if the `if`-body is executed, then the `else`-body is not executed, therefore the `jal`-instruction was emitted. When looking at the code, it should be clear that a merge is only possible after the `if-else`-statement and exactly that location is found with this strategy by looking at the last instruction before the target location of the `beq`-instruction.

Now, let us take a look again at the translation of a `while`-loop to RISC-U instructions.

```
// example snippet
int example() {
  ...
  while (a < 42) {
    a = a + 1;
  }
  // something
  ...
}
```

```
ld t0,-8(s0)
addi t1,zero,42
sltu t0,t0,t1
beq t0,zero,6    // check condition
ld t0,-8(s0)     // loop body
addi t1,zero,1   // loop body
add t0,t0,t1     // loop body
sd t0,-8(s0)     // loop body
jal zero,-8      // jump back to
                 // loop condition
ld t0,-8(s0)     // outside of loop body
addi t1,zero,1
sub t0,t0,t1
```

Figure 3.3: RISC-U Translation

Again, the `beq`-instruction represents the check of the (loop) condition. The last instruction before the target location of the `beq`-instruction is a `jal`-instruction with a negative immediate value, which represents the jump back to the loop condition. That is how `while`-loops work, after the loop body was executed, the loop condition is checked again in order to determine whether to execute the loop body again or not. In that case, the merge is possible after the loop body.

Also, recursion has to be taken care of and that is a bit more complicated: two contexts cannot be merged if they are at different recursion levels. So, it is possible that two contexts are at the "same" instruction, but are actually at different recursion levels. That could be the case if one context is deeper inside the recursion than the other context. Currently, it is implemented in such a way that contexts are not merged if they are inside a recursion. After the context is outside the recursion, it can be merged again. Each context stores information about whether it is in a recursion or not. If a context calls a method which the context has already called and not finished, then the context enters a recursion. So, a context also needs to store the methods it has called and not yet finished. That is not the most efficient and elegant way, since contexts could potentially be merged earlier, as long as they are on the same level of recursion. However, the merge is still working that way and the difference should be not that much in most of the cases.

It is very important to note that this strategy of determining the merge location is highly dependent on the selfie compiler. When looking at the selfie compiler, one can clearly see how it translates `if-else`-statements and `while`-loops to RISC-U code and therefore this strategy works.

3.4.2 Execution of Contexts

As already mentioned earlier, it is very important to understand that contexts (paths) can only be merged at suitable locations. It is not possible to just randomly merge two contexts, they have to be at the same program location for the merge to be possible. Of course, in order to be merged, both contexts have to be stopped from execution. It is quite a complex task to handle the execution of all the different contexts in the right way, since there are many pitfalls: for example, if one context is not stopped at the merge location and is executed further, it cannot be merged with a context which has correctly stopped executing at the merge location. That means, contexts have to be executed in the right order and they also need to be stopped from executing at the right location. Then, suitable contexts can be merged and in the end, every context, whether it was merged or not, needs to finish its execution.

For handling the execution of all these contexts, a state machine around contexts was implemented. This state machine represents the handling of the contexts in detail and is in principle based on the algorithm for generic symbolic exploration, which was introduced earlier. It represents the function for picking the next state (context) [10]. In order to understand better how that works, let us take a look at the following illustration.

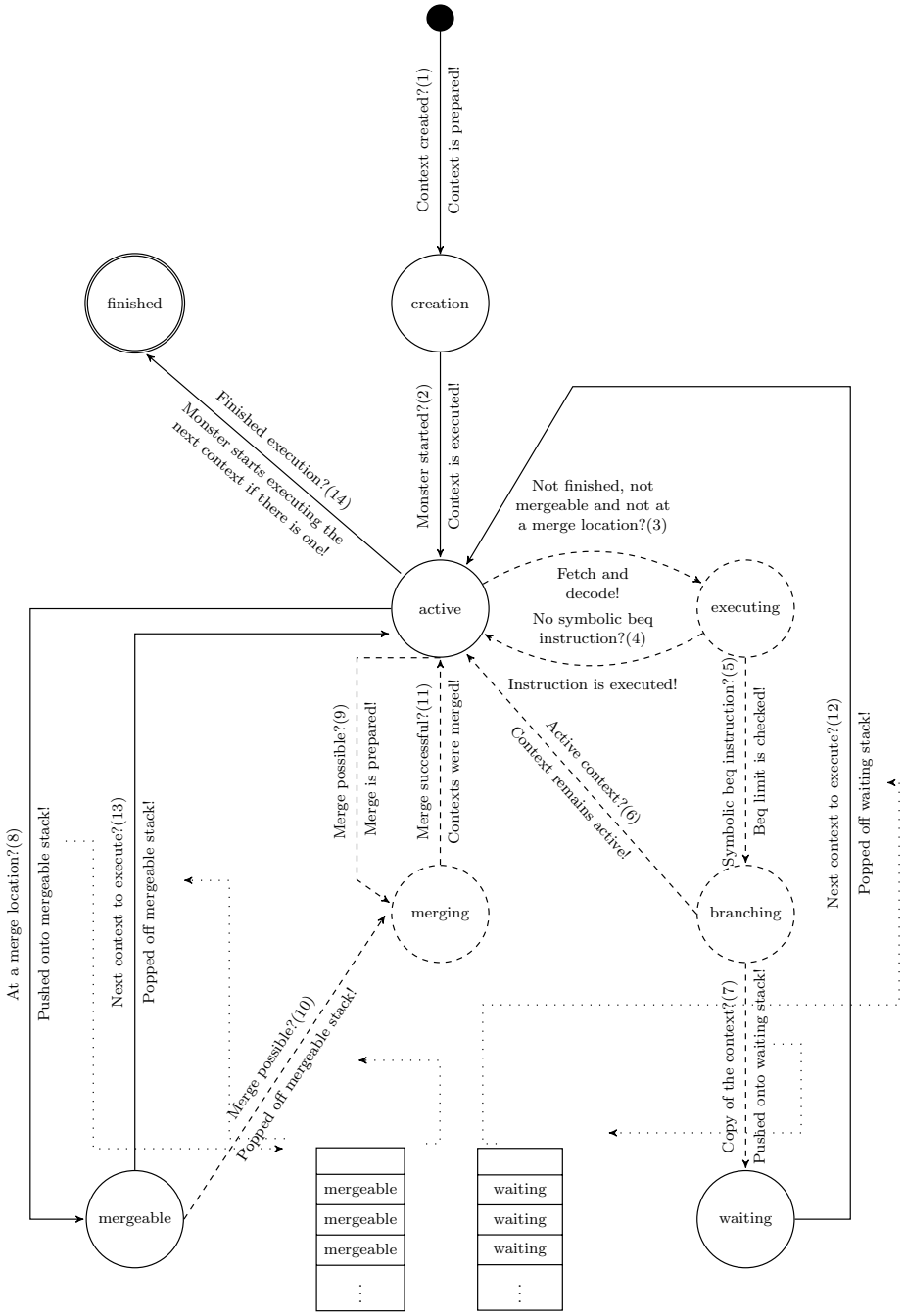

Figure 3.4: State Machine of the Hybrid Symbolic Execution and Bounded Model Checking Engine

In the beginning, a context is created (1) and is then given to the engine to be executed (2), that just represents the start of the program. This context is running on monster and is active. Basically, a context can be active, inactive or finished, but more on that later. Only one context can be active at a time. If the active context is not finished, cannot be merged and has not reached its merge location, then the next instruction is fetched and decoded (3). At this point, the context is executing. Now, there is a difference whether the decoded instruction is a `beq`-instruction with symbolic values or not. If not, the instruction is simply executed and the context is active again (4). If the decoded instruction is a `beq`-instruction with symbolic values, then the context is branching (still active) and the `beq`-limit is checked (5). The `beq`-limit is an upper bound on the maximal number of `beq`-instructions at which a new context is created, as explained before. If the `beq`-limit has not been reached yet, a new context is created which is basically a copy of the active context (except for the program counter and the path condition). Now, the transition depends on whether we consider the active context or the copy of the active context. The copy of the active context is pushed onto the stack of waiting contexts (7). This context is waiting, which means inactive. On the other hand, the active context remains active (6).

If the active context reaches its merge location (8), the context is mergeable (inactive) and is pushed onto the stack of mergeable contexts and monster starts executing another context. Also, a merge could be possible for the active context (9). In that case, the corresponding mergeable context is popped off the stack of mergeable contexts (10) and both the active and the mergeable context are merging. The mergeable context is merged "into" the active context and the active context stays active after the merge (11).

A waiting context can be popped off the stack of waiting contexts in order to become active (12), if it was chosen by monster as the next context to be executed. That choice depends on various factors. The same is true for a mergeable context which can be popped off the stack of mergeable contexts, if it was chosen by monster to be executed (13). When a context stops executing for whatever reason (finished the program, encountered some error, reached some limit, etc.), then it is finished and monster needs to choose the next context to be executed (14), if there is any.

3.4.3 Actual Merge

While it is clear now how the merge location is determined and how the execution of the different contexts is handled, the question how the actual merge works still remains. Up until now, we have just assumed that we can somehow magically merge two different contexts in such a way that everything works correctly afterwards. It is time to demystify that. In order to do that, we first have to understand in what ways contexts can differ from each other.

Different contexts may have different

- execution depths & `beq`-limits
- path conditions
- register values
- (symbolic) memory stores

All of these have to be merged in the correct way.

Execution Depth & Beq-Limit

The execution depth represents the number of instructions a context has already executed. That is important in order to stop a context from exceeding the limit set by the maximal execution depth. Merging the execution depth of two contexts is quite simple: just assume the bigger one. If we do it that way, we may overestimate the execution depth, but we never underestimate it, which is the important thing here.

The `beq`-limit can also differ, but here the approach is a bit different. The mergeable context is always merged into the active context and the active context remains active. Even if the `beq`-counters are different, the `beq`-counter of the active context is not modified. This is important for the exploration of the program. If the mergeable context has already reached its limit and the counter of the active context would be modified, then afterwards no new paths would be explored anymore.

Path Condition

Each context has a specific path condition which basically stores all the constraints of the path. For example, the path condition is different, depending on whether the path of the `if`-body or the path of the `else`-body was executed. Merging the path conditions of two different contexts is quite simple too: we use a disjunction of the two path conditions. So, if pc' is the path condition of the active context (before the merge) and pc'' is the path condition of the mergeable context, then the new path condition pc of the active context after the merge would be $pc = pc' \vee pc''$ [10].

Register Values

As already mentioned before, for each path the full machine state is defined. Each context has its own registers, whose values can obviously differ from the values of the registers of another context. It is important to note here that the values of the registers can either be concrete or symbolic. This is not a surprise, as the symbolic execution of a program involves symbolic and concrete values. If a concrete value is loaded into a register, then the value of the register is concrete. However, it is also possible that a symbolic value is loaded into a register, which would mean that the value of the register is symbolic.

Let us recall that in our engine symbolic values are represented as strings written in the SMT-LIB language. The SMT-LIB language defines an useful expression that looks like this:

```
(ite pc a b)
```

The `ite` stands for "if then else". So, if `pc` is true, then use `a`, otherwise use `b`. That is exactly what we need. For each register of the active context, we look at the corresponding register of the mergeable context. Then, we use the path condition of the active context as the condition in this `ite` expression and the values of the registers as the remaining two arguments. We then update the value of the register of the active context with this string representing the `ite` expression. This is done for each register, unless the value in the registers is the same in both contexts.

This may be a bit hard to grasp, but let us look at it that way: we have two contexts with two different path conditions and different values in their registers. One context - the mergeable context - is merged into the other context. However, we must not lose the information of the mergeable context. If `pcActive` is the path condition of the active context, `valActive` the value in the given register of the active context and `valMergeable` the value in the given register of the mergeable context, then the `ite` expression would look like this:

```
(ite pcActive valActive valMergeable)
```

If we write this string into the given register of the active context, then that would represent the following check for the solver: if the path condition of the active context is true, use the value of the active context. If the path condition of the active context is not true (and therefore the path condition of the mergeable context is true), use the value of the mergeable context.

That way we do not lose the information about the context which is merged into another context.

Symbolic Store

Merging the symbolic store, which can be seen as the memory of a symbolic context, is quite complex. Before explaining how to do that, we first need to understand what symbolic stores are in the first place. In essence, the symbolic store maps variables to either concrete or symbolic values. More precisely, it maps (virtual) addresses to values. To say it a bit simplified, in the symbolic store the (concrete or symbolic) values of variables are stored and can be accessed via the (virtual) address of the given variable. So, we could consider the symbolic store as the memory of the context. However, the implementation of symbolic stores in the engine is a bit special: the store is a singly-linked list, where each entry of the list is a symbolic memory word.

A visualization of a symbolic memory word looks like this:

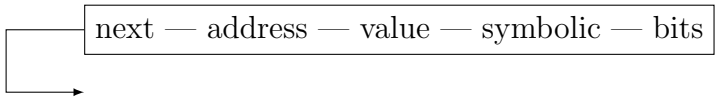

Figure 3.5: Symbolic Memory Word

The symbolic memory word consists of five entries. The first entry is a pointer to the next symbolic memory word. The second entry is the (virtual) memory address. The third entry is the concrete value. The fourth entry is the symbolic value and the fifth entry is the number of bits of the bit vector.

The symbolic store is a singly-linked list of such symbolic memory words. An abstract illustration of (a part of) the symbolic store would look as follows:

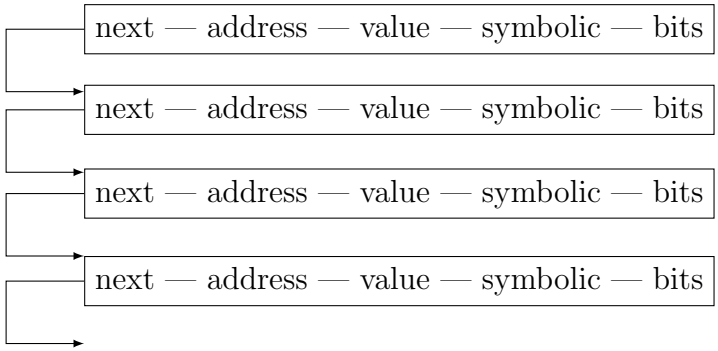

Figure 3.6: Symbolic Store

Storing new entries in the symbolic store works like this: if the virtual address of the given entry is not yet in the symbolic store, then it can be added to the list by creating a new symbolic memory word with a pointer to the latest symbolic memory word of the list. However, if the virtual address of the given entry already exists in the symbolic store, then the corresponding symbolic memory word is updated and no new symbolic memory word is created. Of course, that means that for every insertion the whole symbolic store has to be searched in the worst case, in order to make sure that the virtual address does not already exist. In other words, the complexity for inserting

is linear in the size of the symbolic store. The complexity for loading stores is also linear in the size of the symbolic store, since in the worst case the whole symbolic store has to be searched for the given virtual address. One might wonder why we do not just create a new symbolic memory word for each new insertion into the symbolic store, which would give us a constant complexity for inserting. We chose that way because of the merging, which will be outlined a bit later.

It is also important to understand how the management of these symbolic stores works. At first, it was implemented in such a way that every context had its own symbolic store. So, when a context was created, which basically means being copied from another context, the whole symbolic store was also copied. It is not hard to imagine that this approach was not quite efficient, since there are a lot of different contexts and normally the symbolic store grows with the execution depth. The advantage, however, was that the merging of two different symbolic stores was quite easy: just look for each entry in the symbolic store of the active context if there is a corresponding entry in the symbolic store of the mergeable context and if so, use an `ite` expression - as explained before - and update the value of the entry of the active context with that string.

While this approach made the merging a bit easier, it was obviously not very efficient and as programs tend to get quite big, this could have lead to a serious bottleneck in performance. Thus, a different approach had to be found. The way it works now, is a bit different: each context has a symbolic store, where a portion of this store is unshared and the rest is shared. We could call this partially shared symbolic stores.

At the very first beginning, when there is just one context, all of its symbolic store is unshared, of course. If the context encounters a `beq`-instruction with symbolic values and therefore creates a new context, the symbolic store is not copied. Instead, a special entry is inserted into the symbolic store marking that the symbolic store from this entry "downward" is shared. Both contexts have access to this portion of the symbolic store, therefore it is shared. However, from this entry "onward" both contexts also have a unshared symbolic store. Every new entry is inserted into the corresponding unshared symbolic store, meaning that every new entry is now exclusive to the corresponding context. Both unshared symbolic stores contain a pointer to the

shared symbolic store. The context knows when the shared symbolic store begins because of the special entry that was inserted.

With this approach, the symbolic store is never copied, which is a huge improvement in terms of efficiency. Before, creating a new context meant copying the whole symbolic store of the active context. The complexity was linear in the size of the symbolic store. Now, just one new entry is inserted. Although we have said before, that the complexity for inserting is linear in the size of the symbolic memory, this is not true in this case, since it is a special entry. This entry does not represent a specific variable and therefore no check is necessary, whether the given entry already exists in the store. Hence, the entry is just appended to the list, giving us constant complexity.

This concept of partially shared symbolic stores is probably rather difficult to understand. An example might make it a bit more understandable. So, let us take a look at an example, where we also see how the merging of two symbolic stores of two different contexts works. However, it is not absolutely necessary to understand the handling and merging of the partially shared symbolic stores in detail, so the following explanation will be a bit simplified, as a detailed explanation would go beyond the scope of this thesis.

Let us look at the very first context. The following (simplified) stack represents the symbolic store of that context.

Figure 3.7: (Simplified) Symbolic Store

Then, this context encounters a `beq`-instruction with symbolic values, which causes the creation of a new context. As mentioned before, from this point onward both contexts have a symbolic store with shared and unshared portions. The symbolic store represented in Figure 3.7 is shared between both contexts. The two new stacks represent the unshared symbolic stores. Both unshared symbolic stores contain an entry, which points to the shared symbolic store and indicates the beginning of the shared symbolic store. That would look like this:

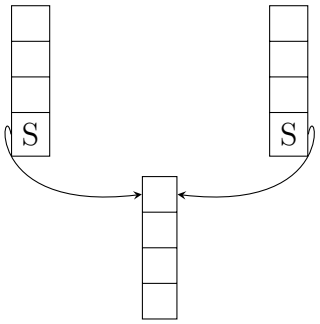

Figure 3.8: Unshared Symbolic Stores

Now, let us assume that the symbolic stores of these two contexts are merged. For example, the right symbolic store could be the symbolic store of the mergeable context and the left symbolic store could be the symbolic store of the active context. The most important thing here is that both contexts know exactly when their unshared symbolic stores end. What happens now is a bit complicated: for each entry of the unshared symbolic store of the active context, the whole symbolic store of the mergeable context is searched in order to merge values with the same address via an `ite` expression and overwrite the value in the symbolic store of the active context with this `ite` expression. If a value is merged, then it is marked as "merged" in the symbolic store of the mergeable context, since it should not be merged again. Not just the unshared symbolic store of the mergeable context is searched, but the whole symbolic store, so also the shared symbolic store is searched. After that has been done for each entry of the unshared symbolic store of the active context, the same is done from "the other side", i.e. for each entry of the unshared symbolic store of the mergeable context. If we are merging "from the side" of the mergeable context, we also need to know how far we are into the shared symbolic store. If we are too far into the shared symbolic

store, we must not overwrite the value, but insert it into the unshared symbolic store of the active context. We always write the `ite` expression into the symbolic store of the active context, not into the symbolic store of the mergeable context.

After the merge is finished, it would look like this:

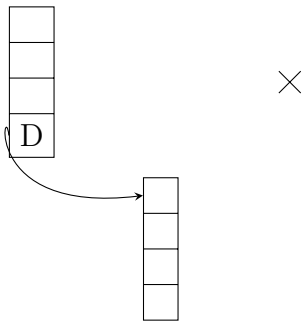

Figure 3.9: Unshared Symbolic Store

Note that the entry indicating the beginning of the shared symbolic store was modified, since this portion of the symbolic store is not shared anymore after the merge. This example could be illustrated further. The important thing to understand here is that it is absolutely essential that every two contexts which are going to be merged, need to know exactly where "their" shared symbolic store begins. This beginning of the shared symbolic store differs in every pairs of contexts, so they need to know that and there cannot be any differences. That is also the reason why, if a context suddenly exits (due to an error or anything else), the special entry in the symbolic store of the corresponding context needs to be updated.

The complexity of merging symbolic stores that way is quadratic in the size of the symbolic stores of the two merged contexts. That means, the complexity of merging two contexts, as it is implemented in the hybrid version, is also quadratic in the size of the symbolic stores of the two merged contexts.

As already mentioned, that explanation was a bit simplified and maybe hard to understand. It is not necessary to understand everything in detail about the partially shared symbolic stores. We just need to remember that the symbolic stores have to be merged as well, and that is done via the `ite`

expression defined in the SMT-LIB language.

3.5 Detecting Division by Zero

Another goal was to automatically detect divisions by zero, since they are undesired in general. That was actually quite easy to implement. When does a division by zero occur? If a division is made with the divisor being zero. So, all we have to do is to check at each division, if this division is reachable and if the divisor can be zero. If yes, a division by zero is possible. That means, at each `divu`-instruction it is checked whether the instruction is reachable, i.e. the path condition is satisfiable, and if the divisor can be zero. That is simply a conjunction between the path condition and the assertion that the divisor is zero. If that conjunction is satisfiable, then the solver can calculate a concrete input which would lead to the division by zero.

3.6 Detecting Invalid Memory Access

Detecting invalid memory access works almost the same as detecting a division by zero. If memory access happens, checking whether that location is reachable and whether the (virtual) memory address is invalid suffices to detect invalid memory access. Again, the satisfiability of the conjunction between the path condition and the assertion that the (virtual) memory address is invalid needs to be checked.

Chapter 4

Experiments

In this chapter the results from experiments with the hybrid engine are presented. The primary focus lays on the comparison of symbolic execution and bounded model checking. In our engine, symbolic execution corresponds to disabling the merging of contexts, and bounded model checking corresponds to enabling the merging of all contexts. Also, the difference in runtime using unshared symbolic stores and partially shared symbolic stores is shown. Test results which measure time are averaged over 100 runs each in order to guarantee reliable results. All experiments ran on a machine (iMac 27-inch, Late 2013) with a 3.4GHz quad-core Intel Core i5 processor (Turbo Boost up to 3.8GHz), 8GB of 1600MHz DDR3 memory, 256 KB L2 Cache (per Core), 6MB L3 cache, and macOS Mojave Version 10.14.5.

Different example programs, which can be viewed on GitHub[1], were used for testing the engine. In essence, almost all of the examples work in a similar way: the input is symbolic and only the input character '1' (binary: 00110001) leads the execution to a return value which is not zero. We defined a constraint that checks for exactly this case which returns a non-zero value. In other words, the goal was to execute the examples symbolically, use Boolector to solve the generated SMT-LIB formulas and to calculate a concrete solution. If everything succeeds, then these concrete solution would be the character '1'.

[1] `https://github.com/cksystemsteaching/selfie/tree/btor2/manuscript/code/symbolic`

The examples represent the following cases:

- simple assignment
- simple `if-else`-statement
- nested `if-else`-statement
- non-nested loop
- two-level nested loop
- three-level nested loop
- recursive factorial
- recursive ackermann
- nested recursion (recursive method calls another recursive method)

Also, two more examples were used in order to test whether the engine would detect a division by zero and illegal memory access. In fact, it automatically did.

In the following experiments, the examples were executed with a `beq`-limit of 35 and an unbounded maximal execution depth, except for the two level-nested loop and three-level nested loop examples using symbolic execution. In that case, these two examples were executed with a maximal execution depth of `300` (instructions), in order to keep the translation time reasonable.

4.1 Size of Generated Files

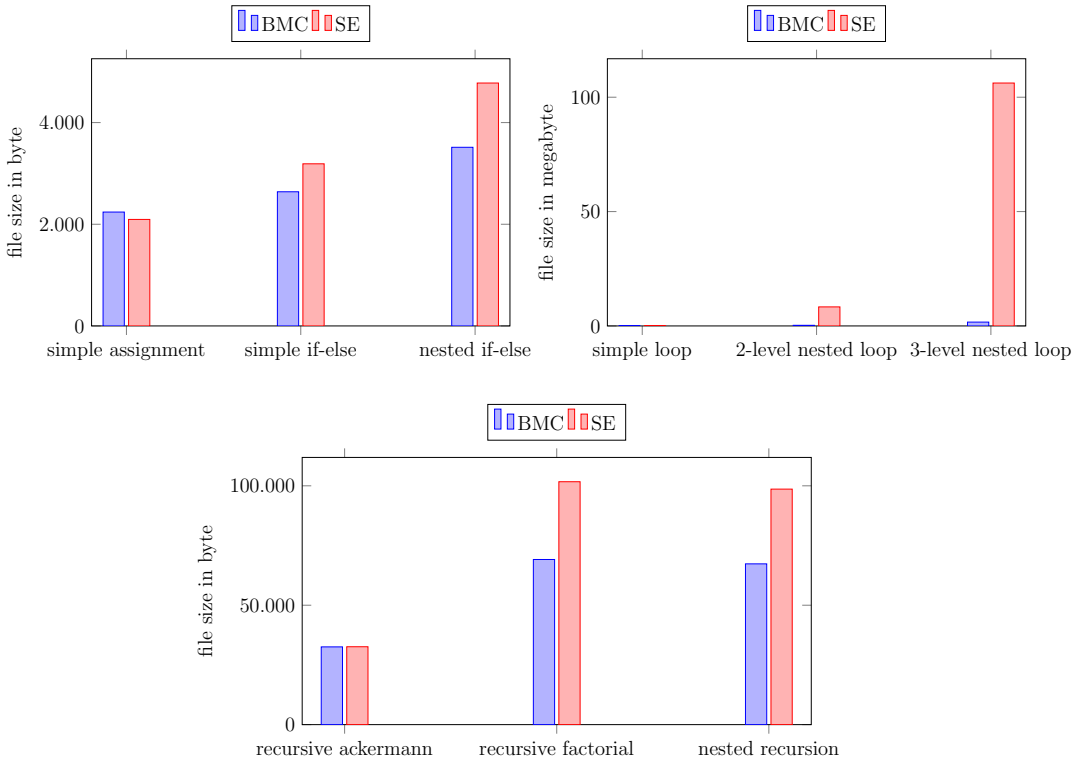

Figure 4.1: Size of Generated Files

It can be seen that the files created by symbolic execution are larger than the files created by bounded model checking. That makes sense, since bounded model checking merges contexts while symbolic execution does not merge contexts. Each context writes at the end of its execution an output formula, which checks whether the path condition is satisfiable and also if the return value is not zero, into the generated file. The difference is especially visible when a lot of contexts are created, as it is the case in the loop examples. The more contexts are created, the more output formulas are produced. Thus, in that regard bounded model checking is clearly preferable over symbolic execution.

4.2 Translation Time

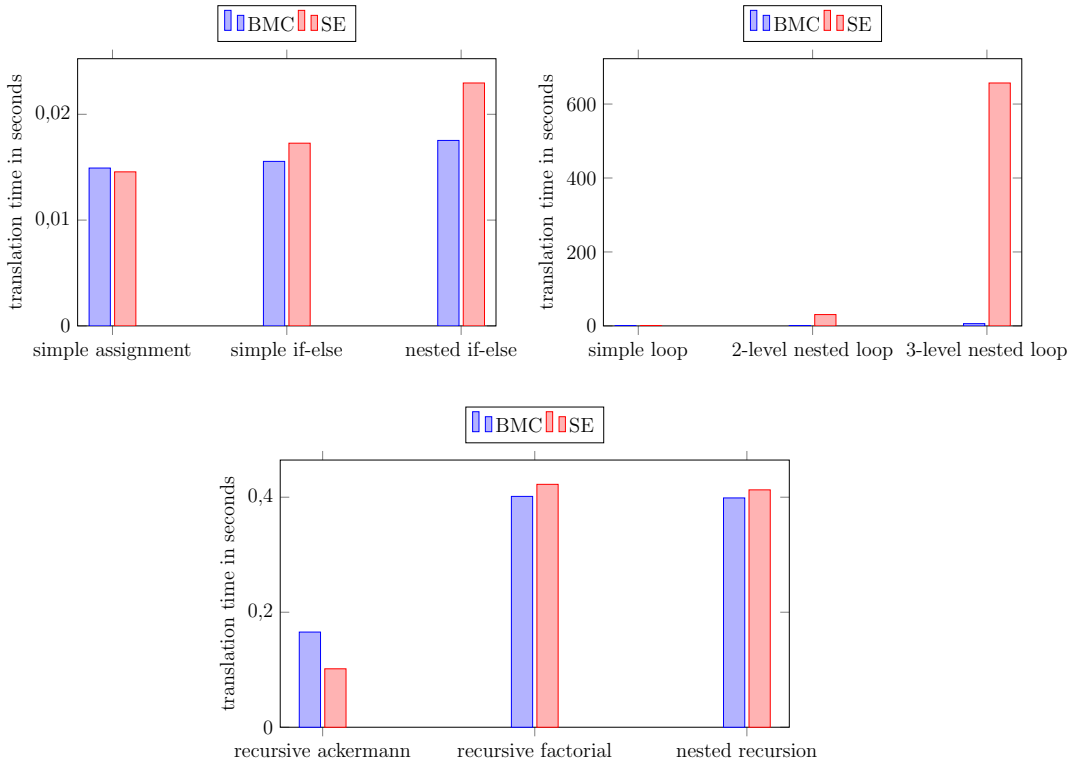

Figure 4.2: Translation Time

These results indicate that regarding the translation time the performance of bounded model checking is better than the performance of symbolic execution in most cases, especially if a lot of contexts are created. Bounded model checking performs a lot better on the loop examples than symbolic execution. That was expected, since the merging of contexts reduces the number of contexts vastly. When looking at the recursive examples, it looks as if symbolic execution is sometimes even better. However, one has to be careful when considering these examples, because the implementation of handling recursion in bounded model checking is not very efficient and therefore it is no surprise that the performance of bounded model checking on the recursive examples is not that good.

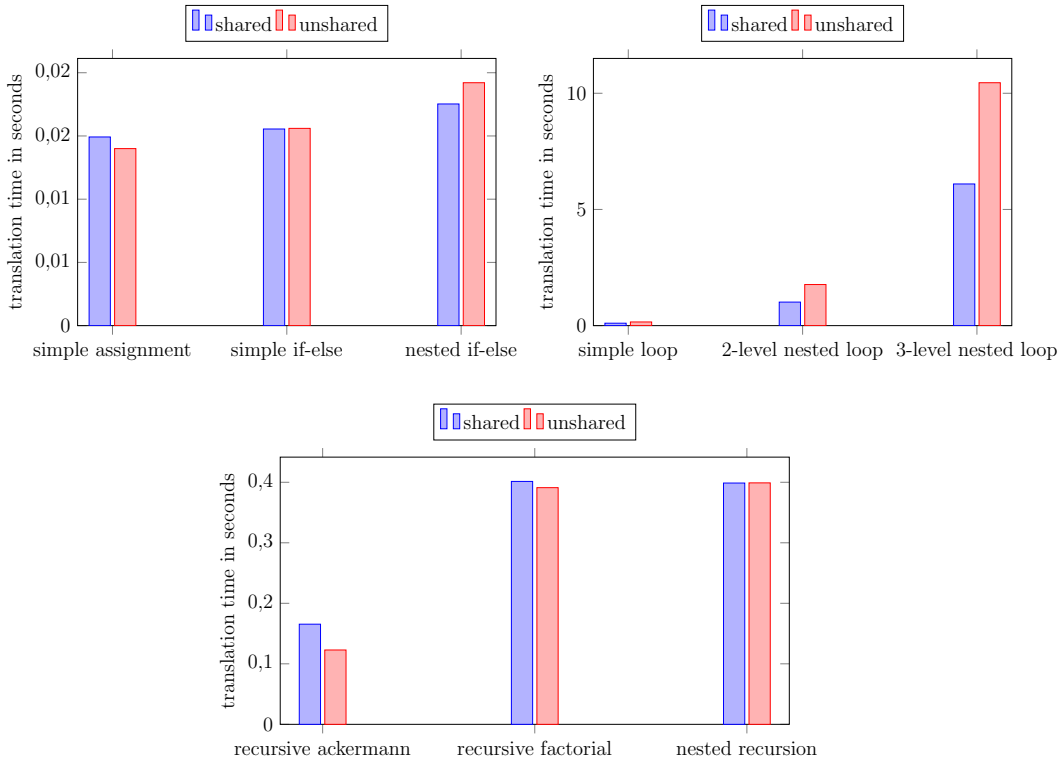

Figure 4.3: Translation Time

As this figure shows, the implementation of (partially) shared symbolic stores made quite a difference in some cases. Regarding the loop examples, the translation time could be reduced by almost half, which is a great improvement. Obviously, partially shared symbolic stores can only show off their advantages when a lot of contexts are created. The difference is not that much if only a few contexts are created. It even seems to be a little bit slower on these examples with only a few contexts, that may be due to the increased complexity of the merging. But that is not a big problem, since normally programs are not that small and tend to have a lot of paths. Implementing partially shared symbolic stores was quite a complex task, however, as the results imply, it was worth the work.

4.3 Time to Show Satisfiability

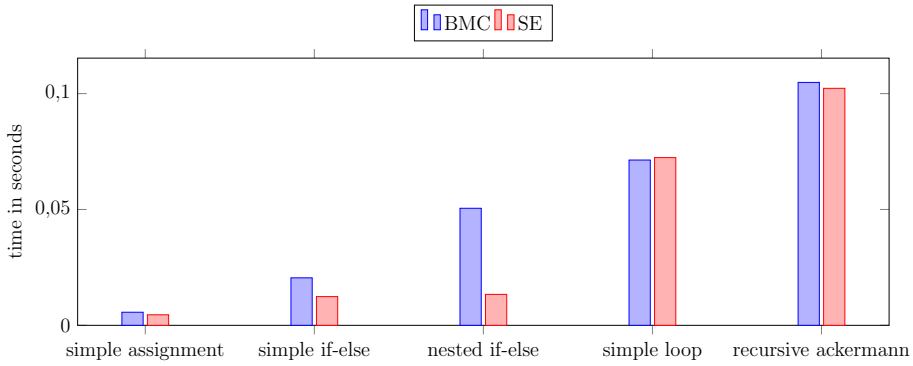

Figure 4.4: Time to Show Satisfiability

These results present the time it took Boolector to solve the generated SMT-LIB files. However, these results are a bit special and should only be considered with caution. Normally, one would expect that the formulas created by bounded model checking would take longer to solve. The reason for that is that with bounded model checking the formulas tend to get quite big, as all the contexts (and their formulas) are merged together (into one big formula). This effect can be seen in the first three examples. However, since with symbolic execution every context produces a formula at the end of its execution, the SMT-LIB files generated by symbolic execution consist of a lot of (small) formulas which all have to be solved, one after another. So, it seems like it takes Boolector quite some time to solve a lot of different small formulas. Also, Boolector prints the solution for every formula. Printing this output could be some kind of bottleneck. These are the reasons why the other examples were not considered for these tests and the results should only be used with caution.

Chapter 5

Conclusion

To sum up, the motivation behind this thesis should be clear now. We are very dependent on software and it is in our best interest to ensure the correctness of that software. Therefore, the methods symbolic execution and bounded model checking were explained and the great differences between these two methods were outlined. Also, a detailed explanation of the hybrid engine in Selfie was given in such a way that one should not only be able to understand the functionality of the engine, but also be able to understand how the principles work. It was explained in detail how the symbolic execution engine had been extended into the hybrid engine by implementing the merging of contexts. The test results obtained from this engine showed that both of the methods have advantages and disadvantages. In other words, we cannot say that one method is better than the other in general, only in certain situations one method may be preferred. For example, if path explosion is a problem, then bounded model checking should be chosen over symbolic execution. On the other hand, if the solver is the bottleneck, symbolic execution may be preferred.

Regarding the engine, there are some aspects which can be improved in the future. The first thing that comes to mind would be the handling of recursion. A more efficient way should be implemented, which does allow merging inside of a recursion. Another feature would be the implementation of some properly justified merging heuristics, i.e. disabling/enabling the merging under some circumstances automatically. Furthermore, it may be possible to simplify the path conditions in some cases. What is more, additional checks could be implemented. Also, more extensive tests could give an even better

idea of when to use symbolic execution or bounded model checking. Another amazing goal would be to use this engine on the whole Selfie project. However, these are all ideas for the future.

Bibliography

[1] Jeff Desjardins. *How Many Millions of Lines of Code Does It Take?* 2017. URL: https://www.visualcapitalist.com/millions-lines-of-code/ (visited on 07/29/2019).

[2] Aarian Marshall. *What Boeing's 737 MAX Has to Do With Cars: Software*. 2019. URL: https://www.wired.com/story/boeings-737-max-cars-software/ (visited on 07/29/2019).

[3] Christoph Kirsch. *Selfie: Computer Science for Everyone*. https://github.com/cksystemsteaching/selfie/blob/master/manuscript/state.md.

[4] James C. King. "A New Approach to Program Testing". In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 228–233. ISSN: 0362-1340. DOI: 10.1145/390016.808444. URL: http://doi.acm.org/10.1145/390016.808444.

[5] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: http://doi.acm.org/10.1145/360248.360252.

[6] Armin Biere et al. "Symbolic Model Checking Without BDDs". In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 193–207. ISBN: 3-540-65703-7. URL: http://dl.acm.org/citation.cfm?id=646483.691738.

[7] Armin Biere et al. *Bounded Model Checking*. 2003.

[8] Thoms Ball. "The Concept of Dynamic Analysis". In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 216–234. ISSN: 0163-5948. DOI: 10.1145/318774.318944. URL: http://doi.acm.org/10.1145/318774.318944.

[9] Edsger W. Dijkstra. "ACM Turing Award Lectures". In: New York, NY, USA: ACM, 2007. Chap. The Humble Programmer. ISBN: 978-1-4503-1049-9. DOI: 10.1145/1283920.1283927. URL: http://doi.acm.org/10.1145/1283920.1283927.

[10] Volodymyr Kuznetsov et al. "Efficient State Merging in Symbolic Execution". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 193–204. ISSN: 0362-1340. DOI: 10.1145/2345156.2254088. URL: http://doi.acm.org/10.1145/2345156.2254088.

[11] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[12] Alireza S Abyaneh et al. "Selfie: Towards Minimal Symbolic Execution". In: *MoreVMs 2018*. Vol. 39. 9. Nice, France, Apr. 2018, pp. 70 –77. URL: https://hal.archives-ouvertes.fr/hal-01852071.

[13] Christoph M. Kirsch. "Selfie and the Basics". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: ACM, 2017, pp. 198–213. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133857. URL: http://doi.acm.org/10.1145/3133850.3133857.

[14] Christoph Kirsch and Sara Seidl. *Selfie: Introduction to the Implementation of Programming Languages, Operating Systems, and Processor Architecture*. 2019. URL: http://selfie.cs.uni-salzburg.at/slides/ (visited on 08/17/2019).

[15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[16] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0 system description". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), pp. 53–58.