Generating Path Conditions for Bounded Model Checking of RISC-V Code in Selfie

by

Sebastian Landl
Student ID: 01524483

Submitted to the Department of Computer Science
University of Salzburg
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

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

August 2020

Abstract

Given a 64-bit RISC-V binary we compute a single logical formula that is satisfiable if and only if there exists input to the code such that there is a division by zero, invalid memory access, or a non-zero exit code when executing no more than a given maximum number of any type of instructions and a given maximum number of branch instructions. We do this by creating a logical formula, which is satisfiable if and only if a given instruction is reachable, called a path condition for each possible path of the program. This is commonly known as symbolic execution. We then merge different path conditions as soon as there is an instruction which can be reached from different paths in the control flow. Creating a single formula for all paths mimics bounded model checking. This is done by checking after every single executed instruction, whether there is another path condition which can be merged with the one we just extended by executing one instruction. Two path conditions are mergeable if they describe the reachability of the very same instruction and if the call stacks of their respective execution paths are the same. For that reason the call stacks of all paths are stored in a global tree structure. To determine which instruction shall be executed next, i.e. which path should be explored further, the path which has the biggest call stack and describes the reachability of the instruction with the lowest program counter is chosen, so that a merge always happens as soon as possible and no possible merge location is ever missed. The time it takes to schedule the next path is linear in the number of paths that are being explored. The size of the generated formula is in $\mathrm{O}(a \cdot 2^b)$ where a is the maximum number of instructions executed on each path and $b \leq a$ is the maximum number of branch instructions for which both branches are considered.

Contents

Chapter 1

Introduction

Given a 64-bit RISC-V binary we compute a single logical formula in the SMT-LIB 2.0 format that is satisfiable if and only if there exists an input such that there is a division by zero, invalid memory access or a non-zero exit code when executing no more than a given maximum number of any type of instructions and a given maximum number of branch instructions. This is done by creating a path condition which is a logical formula that is satisfiable if and only if a given instruction is reachable. Creating such a formula for every branch is commonly known as symbolic execution. Creating a single formula for all branches like we do it here mimics bounded model checking.

What we want to do is to first enumerate the path conditions for all possible paths. Then, whenever there is an instruction which can be reached from different locations in the control flow we want to merge the two path conditions corresponding to that instruction, which we do by combining the two formulae with a logical `or`. This is always done as soon as possible.

In order to achieve this eager merging of paths the call stack for each path is stored in a global tree structure. Then, after every executed instruction on a given path it is checked, whether there is any other path condition which encodes the reachability of the exact same instruction and has the same call stack. If there is such a path condition these two will be merged. Another important detail is how to decide which path to further explore, i.e. on which path the next instruction should be executed. This is done by choosing the path with the biggest call stack and the lowest program counter of the instruction that is encoded by the path condition. This ensures that no possible merge location of two different paths is ever missed, which could happen if two paths could be merged at a certain location, but one of them already executed

instructions beyond that point, while the other path still has not reached that location. The time the algorithm takes to schedule the next path is linear in the number of paths that are being explored. The size of the generated formula is in $O(a \cdot 2^b)$ where a is the maximum number of instructions executed on each path and $b \leq a$ is the maximum number of branch instructions for which both branches are considered. All this is implemented in the selfie system, which is an educational piece of software that works with a C*, which is a tiny subset of C and a tiny subset containing 14 instructions of RISC-V called RISC-U.

1.1 Basics of Program Execution

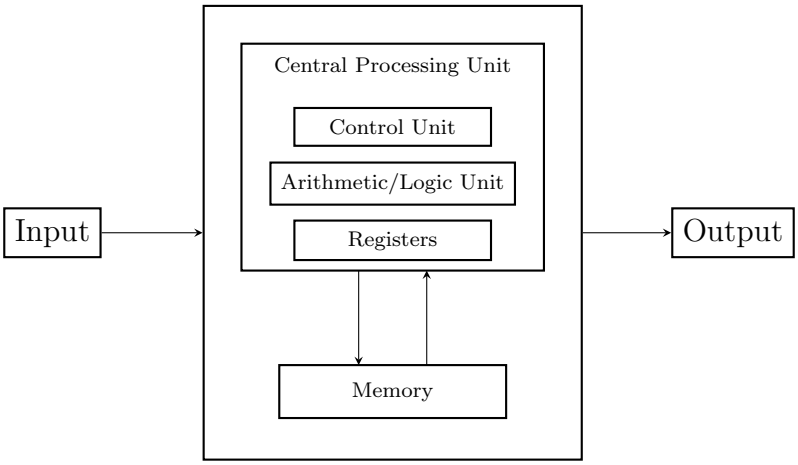

Figure 1.1: Von Neumann Model

We will start with the Von Neumann model (figure 1.1), which is a simple model of a modern computer. It comprises the CPU, memory, input and output.[10][7] We need not focus on the details of the Central Processing Unit, all we require is a basic understanding of what happens when an instruction is executed. In order to explain that the term *machine state* will be introduced first. In this thesis this term describes all values of all bits in a machine. All bits are the entire memory, as well as all registers of the CPU and these bits can be either 0 or 1. In essence program execution is simply the process of fetching an instruction from memory, decoding it and executing it. Executing an instruction means that the machine state changes according to that instruction. And by executing instruction after instruction in the fetch, decode, execute cycle a program is being executed, while it can take input into account and produce output.

1.2 Symbolic Execution

Symbolic execution is a form of program validation. That is to say, it can be used to find errors in a program. In order to do this we first need to establish what an error is. An error could be a division by zero or the wrong output of a program for a given input. An error is simply what we define as an error. We divide all possible machine states into two categories: good and bad machine states. Whenever the machine reaches a bad state an error has occurred.

Of course symbolic execution is not the only to validate a program. A program can also just be executed and afterwards we can check whether any error occurred and whether the output is correct. However, this becomes more difficult when we try to validate programs that work with some form of (user) input, which many programs do. It is common place to define a set of test inputs and then check whether an error occurs and the output is correct, just like before. An experienced programmer can also try to cover all the corner cases, which often are zero or `null` as input, however it will probably not be feasible to cover all possible inputs. We just have to hope that the test cases were chosen well.

This is where symbolic execution as a form of program validation can be very interesting. Instead of only checking a limited amount of test cases the program is executed symbolically. Now what does that mean?

A program contains many choices. In a programming language these can be realized for instance by using an `if`-statement. During the execution of the program the condition is evaluated and the code inside the `if`-statement is either executed or not. This choice can depend entirely on some internal calculation, in which case it would always be the same, or it can depend on some kind of user input. And the result is that only a single path is taken, every time the program is executed. In traditional program validation an effort is made to find test cases such that as many paths as possible are tested. In symbolic execution all paths are explored. And for each path a *path condition* is created, which is a logical formula. And the solution to this formula is the input that allows the program to take this path. We then just have to define what we consider to be an error, i.e. a bad state, in order to execute the program symbolically and get the formula we can solve to get all the inputs that lead to a bad state.[1][5]

A variant of symbolic execution is *bounded model checking*. It addresses a problem of symbolic execution: When exploring a program with many different choices, i.e. paths, the number of paths that need to be explored can get very high, which is called path explosion. In bounded model checking two different paths are merged again in order to contain that problem. A simple example would be an `if`-statement with a symbolic value in its condition. There are two paths, one where the body of the `if`-statement is executed and one where it is skipped, which results in two paths. However, after the body of the `if`-statement the two paths and their path conditions can be merged again.[8]

1.3 The Selfie System

This project was implemented in the selfie system[1], which was created to teach students about programming language design, compilers, concurrency and systems engineering.[6] It comprises:

- a self-compiling compiler
- a self-executing emulator called mipster
- a self-hosting hypervisor called hypster
- a tiny C* library
- a symbolic execution engine
- and much more

At the time the code which this bachelor thesis is about was developed all this features were contained in a single file, which was about 12 thousand lines long. Selfie is written in and compiles C*, which is a tiny subset of C. The instructions that the selfie compiler outputs and mipster interprets are a tiny subset of RISC-V called RISC-U, which contains 14 instructions.[7] The details on how to run selfie and the source code for this project can be found in the git repository. Finally I will simply mention the command to invoke the symbolic execution engine described in this thesis:

```
./selfie -c program.c -se 0 35 --merge-enabled
```

[1] `https://github.com/cksystemsteaching/selfie`

The `-se` option invokes the symbolic execution engine. The arguments `0` and `35` denote the execution depth and `beq`-limit respectively, the meaning of which will be explained in detail in the next chapter. These are just some parameters that need tuning for each program and the values specified here are a good place to start. Finally the `--merge-enabled` option enables the merging of contexts. What exactly that means will be explained in the following chapters.

Chapter 2

Existing Work

The symbolic execution engine described in this thesis was created in collaboration with my colleague Christian Edelmayer, who worked on the engine before in order to compare traditional symbolic execution to bounded model checking. His amazing work is described in his bachelor's thesis "A Hybrid Symbolic Execution and Bounded Model Checking Engine in Selfie". In this chapter I will give a short and simple description of his work, which should be just enough to understand the improvements and changes that are described in this thesis. For a detailed description please refer to his bachelor thesis.[4]

2.1 Monster

The engine is based on something that was already implemented in selfie: Monster, which is a symbolic execution engine for RISC-U. This means it can take a RISC-U binary, symbolically execute it and in turn create a SMT-LIB formula, which is the representation of the logical formula I talked about in section 1.2.[2] In order to understand the contents of this thesis it is not necessary to understand the details of SMT-LIB and neither is it necessary to understand all the implementation details of Monster, however, understanding the basic principles of how Monster works is definitely helpful, which is why this section exists.

Monster is essentially an interpreter for RISC-U instructions. In order to do that it uses the concept of a *context*. A context stores the entire machine state; the contents of all registers as well as the memory. And since monster executes the code symbolically a path condition is stored in the context as well.

A variable is considered to have a symbolic value, when its value is assigned by the `read` system call. When a program is executed normally we use the `read` system call to get user input.

Finally, after we generated an SMT-LIB file with one or more formulae by sybolically executing a program with Monster we then use a SMT-solver to get the input that leads us to a bad state, which is in our case boolector[1].[9]

2.2 Extension of Monster

Monster was then extended by four features; unrolling loops and recursions, merging paths and the detection of divisions by zero as well as invalid memory access, which will be described in the following subsections.

2.2.1 Unrolling Loops and Recursions

In order to explain how loops are dealt with, we should first take a quick look at what a simple loop looks like written with RISC-U instructions.

Listing 2.1: simple-increasing-loop.c

```
1  while (*x < 60) {
2    a = a + 1;
3    *x = *x + 1;
4  }
```

The code is very simple. While `*x` is smaller than 60, `a` and `*x` are increased by one. The corresponding RISC-U assembly code looks like this:

[1] `https://boolector.github.io/`

Listing 2.2: simple-increasing-loop.c

```
1  ld t0,-16(s0)          // line 1
2  ld t0,0(t0)            // line 1
3  addi t1,zero,60        // line 1
4  sltu t0,t0,t1          // line 1
5  beq t0,zero,12[0x1E0]  // line 1
6  ld t0,-8(s0)           // line 2
7  addi t1,zero,1         // line 2
8  add t0,t0,t1           // line 2
9  sd t0,-8(s0)           // line 2
10 ld t0,-16(s0)          // line 3
11 ld t1,-16(s0)          // line 3
12 ld t1,0(t1)            // line 3
13 addi t2,zero,1         // line 3
14 add t1,t1,t2           // line 3
15 sd t1,0(t0)            // line 3
16 jal zero,-15[0x1A0]    // line 4
```

The comments in the assembly code snippet above indicate which line of code snippet 2.1 the individual assembly instructions correspond to.

We assume that `*x` is some kind of user input to the program. What will happen then is that the context executing this piece of code will reach the `beq`-instruction, which compares the values in the registers which are passed as the first two arguments and increases the program counter by the third argument, an immediate value, if they are equal and by one otherwise. At this point another context will be created, by copying the context that originally reached the `beq`-instruction. One context will continue executing the instruction right after the `beq` and the other one will take the branch and jump to address `0x1E0` (in this specific case). This means that both paths are explored by a context and the path conditions are built accordingly. Now we will take a closer look at the context that did not take the branch; this is the context that executes the body of the `while` statement. Eventually this context will reach the `jal`-instruction in line 16 of code snippet 2.2 where it will jump to the beginning of the evaluation of the condition of the `while` statement and reach the same `beq`-instruction again, where another context will be created. In theory this process would never terminate, which would not be desirable at all. There are two mechanisms in place in order to prevent this, which are the maximal execution depth and the `beq`-limit.

The maximal execution depth, which was implemented before Christian Edelmayer worked on the engine, is the maximum number of instructions any context is allowed to execute, which is a way to define how deeply a program may be explored. When this maximum number of instructions is reached the context will simply be terminated.

Another metric of how deeply a program is explored is the number of times a new context is created at a `beq`-instruction, which happens whenever there is a symbolic value involved in the condition. Each context has an internal `beq`-counter and when the `beq`-limit is reached and an `beq`-instruction is encountered where usually a new context would be created this will simply not happen and there will not be a context taking the branch (jumping forward by the number of instructions specified in the immediate value of the `beq`-instruction) for this instruction.

Both of these values are copied every time a new context is created, which ensures that these limits cannot be exceeded by any context no matter when it is created.

2.2.2 Merging Contexts

At the center of the extension of the monster engine was the merging of contexts (and path conditions). The basic idea of the engine is to copy a context when there are two paths that need to be explored, but merge the contexts back together into one context at a later point in time. In order to get an idea of how this whole process works we need to understand the following:

First of all, a location in the binary where both contexts are able to be merged must be identified. On a high level we can say that when we encounter a symbolic condition in a `while` or `if` statement, a new context is created. This new context then proceeds to take a different path, i.e execute different code. However, at a certain point both context will execute the same code again. This would be after the end of a `while` loop, the end of the body of an `if` statement or the corresponding `else` branch, respectively. And since we know exactly how the selfie compiler translates these statements into machine instructions, we can use the following information to calculate the merge location. When we encounter a `beq`-instruction we can determine whether it was part of a `while` loop or an `if` (`else`) statement. To do this we take a look at the instruction right before the jump location specified in the `beq`. There are three options:

- `jal`-instruction with negative immediate value: This indicates that the `beq`-instruction is part of a `while` loop; the jump backwards is used to evaluate the condition again and potentially execute the loop body again to finally arrive at the `jal`-instruction again.
- `jal`-instruction with positive immediate value: This means we are dealing with an `if-else`; the jump forward is used to skip the `else` branch, if the condition is evaluated to `true`.
- Any instruction other than `jal`: The last option is that we are looking at an `if` statement without an `else` branch; the `beq`-instruction simply skips the body of the `if` statement, if the condition evaluates to `false`.

With this knowledge the exact instruction right after the end of a `while` loop or an `if` (`else`) statement can be determined and that location can be set as the merge location. At this merge location contexts and their path conditions that took different branches at the same `beq`-instruction can be merged together again. This was implemented by a special schedule where a context either waits at a given merge location for another context or there is a context waiting for it already in which case they would be merged.

2.2.3 Detection of Bad States

Two new bad states were defined that could be detected by the engine: division by zero and invalid memory access, which both are not desirable.

In order to detect a division by zero, all that needs to be done is to simply check at every `divu`-instruction, whether the divisor can be zero and the path condition is still satisfied. If that is the case an SMT-solver will give us the input for which we reach a division by zero.

Checking for invalid memory access works in a similar way. Whenever a memory access happens two things must be checked: whether the path condition is satisfiable, i.e. the location can be reached, and whether the memory address is invalid. If both of these conditions hold, an SMT-solver will again find the input that leads to this bad state.

Chapter 3

Improving the Engine

This chapter explains the specific problems with the old engine and how the solutions to these problems were implemented. That is the recursion handling that simply did not exist in the old version of the engine. Furthermore, a new schedule of the contexts and a new strategy for merging were implemented, which will be explained in detail in this chapter.

3.1 Recursion Handling/Unrolling recursions

The challenge in implementing proper recursion handling is to identify when two contexts can be merged, as they have to be on the same level of recursion, meaning they both must have the exact same number of recursive calls on their respective call stacks and their call stacks must be the same in general. To achieve this a data structure was implemented to record and quickly compare the call stacks of all contexts and the merging of contexts was changed such that only contexts with the same call stacks will be merged.

3.1.1 Quick Recap of Recursions

Recursion is a simple concept. When a procedure calls itself over and over again until a certain termination condition is true, we are dealing with a recursive procedure. This usually works by having the procedure call itself with a different argument. Then there is for example an `if-else` statement where a branch is taken, depending on the argument the procedure was called with.

One branch will call the procedure again and return the result of that recursive call, while the other branch will simply return a static result. When the terminating branch is taken, that means the termination condition is fulfilled. So when a recursive procedure is called the call stack gets filled up with calls from the same procedure with different arguments. Eventually the termination condition is fulfilled and the procedure returns. This means that this specific procedure call is removed from the call stack and the next procedure can return, since it now has the result from the recursive procedure call. This way all the recursive calls return and the original call can deliver the desired result. This also works with two or more procedures calling each other.[3]

3.1.2 Old Recursion Handling

The general assumption upon which the old strategy for merging contexts relied was that contexts could be merged when they have the same program counter and we can conclude that they are in the same location within the program. However, with recursions this assumptions no longer holds, since two contexts can now have the same program counter, even though they are at different locations in the execution tree of the program, which would mean that they cannot be merged.

The old strategy for handling recursions was very simple. Upon detecting a recursive procedure the merging of contexts would be disabled until after the recursion ends. This is not ideal, as all advantages of merging during the symbolic execution of a program are lost during recursive procedures.

3.1.3 Call Stacks

In order to explain the new strategy for handling recursion, I first have to introduce the concept of a call stack and explain how it is used here.

Whenever a procedure is called, space for it gets allocated on the call stack, i.e. the procedure is pushed onto the call stack. First, if there are any arguments for the called procedure, those will be stored on the stack, so that they can be accessed by the procedure. Also any local variables are stored on the stack as well as the return value of the called procedure, if there is any. After the procedure returns, it is popped from the call stack and now the topmost procedure on the call stack is the caller of the procedure that was just popped from the stack.

Therefore, by examining the call stack, we get information about the history of which procedures have been called and in which order. To use that information a data structure to store the call stacks of each context was needed.

Storing Call Stacks as Linked Lists

The first data structure chosen to store the call stacks (note that here only the procedures are stored and not any arguments or local variables) was a linked list, which works as follows: each context has a pointer to the first element of the linked list and each element of the linked list contains an address of a procedure and a pointer to the next element. Insertion and deletion of an element work in constant time, as we only ever need to perform these operations at the first entry of the list, which means we only need to set a constant number of pointers correctly. Figures 3.1 to 3.2 illustrate the concept of this data structure. In this example `main()` represents the address of the procedure within which another procedure at address `0x0124` is being called, which results in the additional entry in the linked list in figure 3.2.

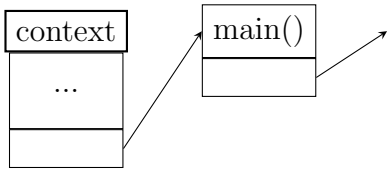

Figure 3.1: Call stack as linked list with one entry

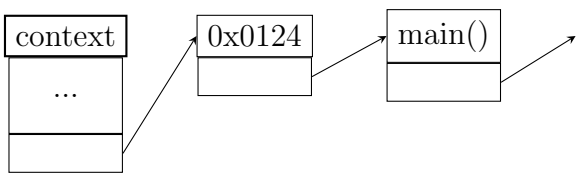

Figure 3.2: Call stack as linked list with two entries

This data structure had two problems: First, each context, the number of which can get quite high, requires space linear in the number of procedure calls to store its call stack. Furthermore, comparing the call stacks of two contexts also takes linear time in length of the shorter call stack. Later on in this section it will be explained why this is important for handling recursions.

Storing Call Stacks as Global Tree Structure

In order to address these problems and massively simplify the comparison of two call stacks a better data structure was needed. The solution we came up with was a global call stack tree (figure 3.3).

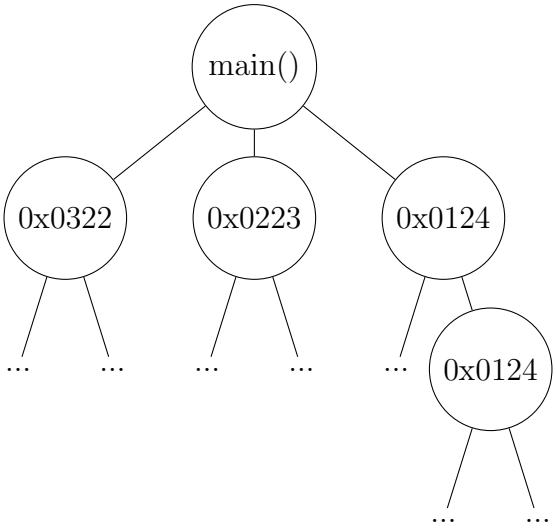

Figure 3.3: Global call stack tree

This tree structure works as follows: The root node contains the address of the `main()` procedure. When a procedure call from within another procedure is being made, a new corresponding child node is created, if it does not exist already. In practice every context stores a pointer to the node in this call stack tree, which corresponds to its call stack (figure 3.4). Upon calling another procedure this pointer is set to the corresponding child node.

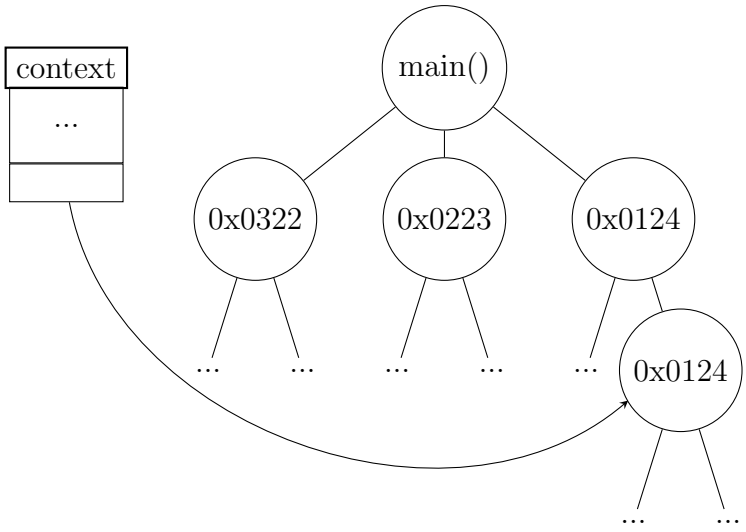

Figure 3.4: Global call stack tree with context

Stepping into a procedure, which takes constant time using linked lists, costs linear time in the number of children of the current node, since we need to check, whether there already exists a child node for the procedure we are stepping into, so we are at a disadvantage regarding asymptotic time complexity. However, since this is a global data structure, as opposed to having each context store their own linked list, we need less space to store the information about all call stacks. Also comparing the call stacks of two contexts only takes constant time as opposed to linear time in length of the shorter call stack, since all we need to do is comparing two pointers. This works because in this data structure every node does not only represent the address of the procedure it stores, but the entire history of the call stack, which can be reconstructed by examining the parent nodes.

After this refinement we encountered one final problem with this data structure. Originally the address that was stored in the nodes of the call stack tree was the address of the procedure in the machine code, i.e. we identified the procedure by the callee. Later on we will examine an example (`recursive-ackemann.c`), where this was a problem, which was easily solved by switching to an identification of a procedure call by caller, meaning instead of storing the address of the actual procedure, the address of where the procedure call happened would be stored, which then not only identifies the procedure, but also the exact location where it was called.

3.1.4 New Recursion Handling

Now that the tools needed to handle recursion properly have been explained, we will take a look at how the recursion handling was implemented.

The old strategy was simple: Whenever a `beq`-instruction was encountered another context was created and they would be merged after they both reached the pre-calculated merge location. At that point they would obviously both have the same program counter and it was therefore (wrongly as we now know) assumed that both contexts must be at the same location within the program and mergeable. However, there is a huge problem with this assumption when it comes to recursion, because when two contexts are inside a recursive procedure, they can have the same program counter even though they are on different levels of recursion, which means that they cannot be merged. In order to remedy this, the following was implemented: Whenever two contexts have reached the same merge location and would normally be merged by the algorithm, their call stacks will be compared first. If they do not have the same length, we know two things. Firstly, the two contexts in question are not at the same location within the program and can therefore not be merged. Secondly, in order for those two contexts to be eventually merged, we have to continue executing the context with the longer call stack and pause the execution of the other context. What we want is for the context with the longer call stack to run into the `beq`-limit, which terminates the exploration of the recursion. Then the recursive calls on top of the call stacks can return which eventually leads to all the contexts created within this recursion to be merged.

3.1.5 Problems with the old merging strategy

Whilst eliminating the problem of merging two contexts that should not be merged due to them having different call stacks, there was still a problem with the pre-calculated merge locations, which can be observed in the following example (code snippets 3.1 and 3.2).

Listing 3.1: recursive-ackemann.c

```c
uint64_t ackermann(uint64_t m, uint64_t n) {
  if (m != 0) {
    if (n != 0)
      return ackermann(m - 1, ackermann(m, n - 1));
    else
      return ackermann(m - 1, 1);
  } else
    return n + 1;
}
```

Code snippet 3.2 shows part of the RISC-U assembly code from `Recursive-Ackermann.c` (3.1), which is simply a recursive C* implementation of the Ackermann function.

Listing 3.2: recursive-ackemann.m

```
ld t0,24(s0)
addi t1,zero,0
sub t0,t1,t0
sltu t0,zero,t0
beq t0,zero,44[0x20C] // outer if
ld t0,16(s0)
addi t1,zero,0
sub t0,t1,t0
sltu t0,zero,t0
beq t0,zero,25[0x1D4] // inner if, split into new contexts

ld t0,24(s0)
addi t1,zero,1
sub t0,t0,t1
addi sp,sp,-8
sd t0,0(sp)
ld t0,24(s0)
addi sp,sp,-8
sd t0,0(sp)
ld t0,16(s0)
addi t1,zero,1
sub t0,t0,t1
addi sp,sp,-8
sd t0,0(sp)
jal ra,-28[0x138]
addi t0,a0,0
addi a0,zero,0
addi sp,sp,-8
sd t0,0(sp)
jal ra,-33[0x138]
addi t0,a0,0
```

```
32 addi a0,zero,0
33 addi a0,t0,0
34 jal zero,21[0x220] // return
35 jal zero,14[0x208]
36
37 ld t0,24(s0)
38 addi t1,zero,1
39 sub t0,t0,t1
40 addi sp,sp,-8
41 sd t0,0(sp)
42 addi t0,zero,1
43 addi sp,sp,-8
44 sd t0,0(sp)
45 jal ra,-47[0x138]
46 addi t0,a0,0
47 addi a0,zero,0
48 addi a0,t0,0
49 jal zero,7[0x220]
50
51 jal zero,6[0x220] // current merge location
52
53 ld t0,16(s0)
54 addi t1,zero,1
55 add t0,t0,t1
56 addi a0,t0,0
57 jal zero,1[0x220]
58
59 addi sp,s0,0
60 ld s0,0(sp)
61 addi sp,sp,8
62 ld ra,0(sp)
63 addi sp,sp,24
```

In order to understand the problem, we need to know a couple of things. An `if` in the source code translates to a `beq`-instruction in the assembly code, where one of the two possible branches is taken. Furthermore, the three `return` statements in the code lead to `jal`-instructions, which change the program counter to the address `0x220`. That is where the epilogue begins, which is a couple of instructions that do some clean up when the procedure returns. Now we will examine the `beq`-instruction at line 10. If the two registers `t0` and `zero` are equal, the program counter will be set to address `0x1D4`, which is line 37. In order to calculate the merge location for the current context and the additional context created at this `beq`-instruction the instruction right before the `beq` is examined and in this case it is a `jal`-instruction, which jumps forward. This results in a merge location at address `0x208` which is in line 51.

What this means is that as soon as both contexts have reached this location, they will be merged, if their call stacks are the same. This is the theory which sounds reasonable, however, in this case there is a problem that renders this merging algorithm non-functional. The last instruction of the `if` branch is in line 34 and the last instruction of the `else` branch is located in line 49. Both of them are `jal`-instructions, which jump to the prologue, which indicates the beginning of a procedure. When we look at the source code we can see that both branches of the inner `if` end with a `return` statement. The result of this is that the merge location in line 51 will never be reached by either of the contexts, which means that they never will be merged.

3.2 The New Merging Strategy and Scheduling of Contexts

In order to address the problems the old strategy of pre-calculating the merge locations and scheduling the contexts accordingly still had even after implementing a way to deal with recursions, a completely new scheduling algorithm was implemented. The idea is quite simple: Same as before at every `beq`-instruction a new context is created, so that both of the branches are taken. Then, after every single executed instruction all contexts are checked for mergeability. And if any two contexts are mergeable, they are merged. As explained previously the conditions for two contexts to be mergeable are:

- They have the same program counter
- They have the same call stack

This leaves the question of which context to schedule after each executed instruction. In order to demonstrate the importance of a correct scheduling of the contexts, let us take a look at a little example. Suppose we have a context A. This context now reaches a `beq`-instruction and another context, B, is created, each context taking one of the two possible branches. For the sake of this example we can simply assume that the `beq`-instruction in question is part of an `if-else`-construct, meaning that one context takes the `if`-branch and the other one takes the `else`-branch. Furthermore we will assume that the contexts can be merged right after the end of the `else`-branch. That means this is the first location, where these two contexts can be merged. In this example that is all we know, which means we do not know anything about the

mergeability of our two contexts after that point. If context A is now executed it will reach the theoretical merge location right after the `else`-branch. Now it has to wait for context B to reach the same location in order for the two contexts to get merged. And since we do not know anything about mergeability from that point onwards, this may be the last location for these two contexts to get merged. In the old algorithm with pre-calculated merge locations, one context would simply wait for the other at that pre-calculated location. Without pre-calculated merge locations we can have the following problem: If one more instruction of context A is executed it may happen that contexts A and B will never be merged. What we need to do is execute the context that is further behind. Which leads us to the next question: How to we determine the context that is furthest behind?
The answer is actually quite simple. The context that must be scheduled next has to fulfill the following two conditions:

1. It must have the lowest program counter
2. It must have the biggest call stack

Now we will examine why these conditions let us schedule the correct context. Intuitively it seems obvious why the first condition is necessary. When a context has a program counter that is lower than the program counter of another context it needs to execute some instructions in order the catch up and reach the location of the other context. However, when we take procedures into account, we cannot make this assumption anymore, since procedures allow contexts to jump around anywhere within the binary. To resolve this problem, we will first take a look at the comparability of the program counter within the same procedure. Obviously, within the same procedure the assumption that a context can catch up to another context with a higher program counter by executing some instructions holds. By combining this knowledge with what we learned from unrolling recursions we get our second condition. It ensures that we always execute the context with the biggest call stack, which is necessary in order to return from recursions. All we have to do then is to compare all contexts which have that (the biggest) call stack and the one with the lowest program counter will be scheduled to execute the next instruction. Then all contexts are checked for mergeability and merged if possible after which the next context is scheduled to execute one instruction. This cycle then repeats over and over until all contexts are merged into one context again and the engine terminates.

3.2.1 Gaining more Independence from the Selfie Compiler

The old merging algorithm relied on the fact that the RISC-U binary that would be symbolically executed was compiled with this very compiler. This was a necessary precondition for being able to detect whether a `beq`-instruction is part of an `if`-condition or a `while`-loop and correctly pre-calculating the merge locations. With the new merging algorithm this dependency no longer exists, as a `beq`-instruction is now simply treated as a point where the program branches and what that instruction means on a higher level is not relevant anymore. Only two dependencies on the compiler remain. It is necessary to know when a procedure call is made and when the contexts returns from that procedure.

As we know that the selfie compiler was used to compile our RICS-U binary, we know how to detect procedure calls. We have to look for a `jal`-instruction, where the destination register is register `ra`, the register which holds return address. Likewise, we can detect when a `return` statement is reached. It is translated to a `jalr`-instruction with the destination register being the zero register, the register provided as argument being register `ra` and the immediate value being zero.

This improvement regarding the general design of the algorithm was not initially a goal of this project. However, it made the symbolic execution engine much more elegant.

3.3 Size of the Generated Formula

Essentially what we are doing is exploring the control flow graph. The maximum depth of our exploration is the maximum execution depth, which is also the upper bound for the length of any path we explore. The control flow graph we explore may have edges such that we can get from a state further down back up to state from before. For the sake of this argument we will rewrite that graph such that it becomes a tree by copying some parts of it and redirecting edges that may have led back up to now lead downwards. When we consider only the part of this tree which we can explore it has a height of the maximum execution depth. In each level of the tree the number of paths can be at most doubled, if each node corresponds to a branch instruction and therefore has

two outgoing edges. The number of times we will take a branch is limited by the `beq`-limit, so we can only have a doubling of paths at most that many times. This gives us the following result: The size of the generated formula is limited by $O(a \cdot 2^b)$, with a being the maximum execution depth and b being the `beq`-limit.

Chapter 4

Experiments

In this chapter the results of several experiments are presented and interpreted. For this purpose the experiments from the first bachelor thesis on the symbolic execution engine[4] were rerun on the same machine the experiments from the new and improved symbolic execution engine described in this thesis were conducted on, in order to allow for a proper comparison between the two. The experiments were run on a dual socket server equipped with two Intel®Xeon®CPU E7-4850 @ 2.00GHz and 125GB of RAM. The following additional software was used:

- Ubuntu server, version 18.04.2 LTS (Bionic Beaver)
- gcc, version 7.3.0
- make, version 4.1
- boolector[1], version 3.2.1

4.1 Translation Time

The translation time is simply the time it takes the symbolic execution engine to generate the SMT-LIB formula(e) for a given program. The programs used here are all very simple and can be found in the selfie repository on GitHub[2].

[1] `https://github.com/Boolector/boolector`
[2] `https://github.com/cksystemsteaching/selfie`

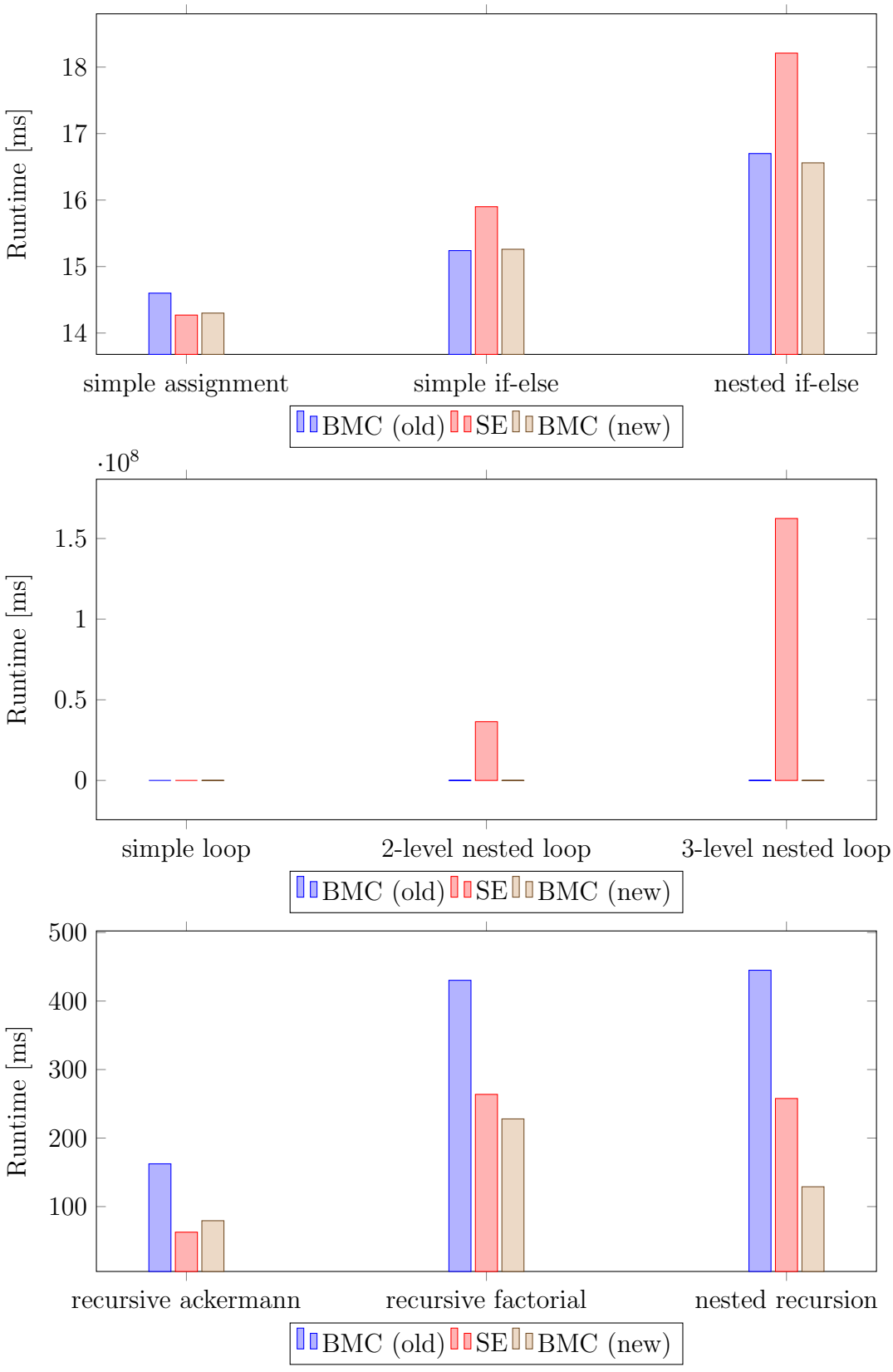

Figure 4.1: Translation time

First we will quickly reiterate the results that were recreated for this bachelor thesis and then examine how the results from the improved engine fit into the picture. The first observation we can make is that in most cases bounded model checking is faster than symbolic execution. This is especially the case in the loop examples, because in these examples lots of contexts are created. And since in bounded model checking all these contexts are being merged again it performs better than symbolic execution. And the higher the number of created contexts the bigger this performance gap becomes. The story looks quite different when examining the recursive examples. We can see that the lack of proper recursion handling leads to longer translation times for bounded model checking when compared to symbolic execution.

Now we will take a closer look at the additional results from the improved symbolic execution engine; the first observation we can make is that the improved engine never performs worse than the old engine, which is really good and not necessarily expected, since after every executed instruction it takes linear time in the number of contexts to schedule the next context. The biggest improvement can be observed when considering the recursive examples. This is probably due to the fact that as opposed to the old engine, which did not implement any proper recursion handling, the new engine handles recursion in a much better way as the contexts are now merged as soon as possible.

4.2 Time to Show Satisfiability

The time to show satisfiability is the time it takes for boolector to solve the generated SMT-LIB formula(e).

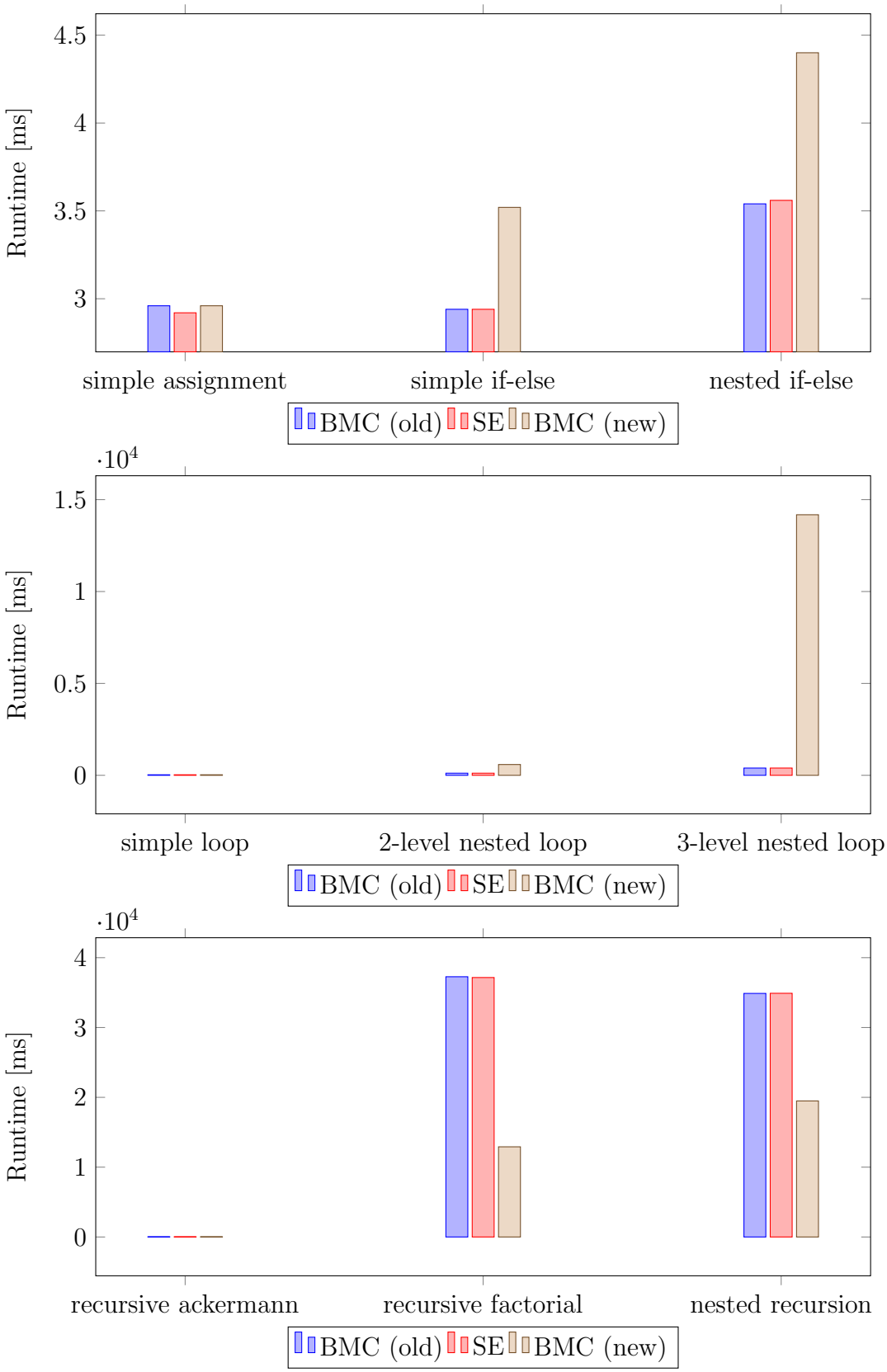

Figure 4.2: Time to Show Satisfiability

Just like with the results from the translation time, these results of how long it takes boolector to solve the produced SMT-LIB formulae were redone and are shown here next to the results of the new engine. The new strategy of always merging as soon as possible will probably produce different formulae. We can be certain it produces a different formula in the recursive ackermann example, as in the old engine not all contexts were merged, however, as the schedule of the contexts has totally changed as well, it would be surprising, if the formula produced would be the same as before. Therefore it is not surprising that there is a difference in how long boolector takes to solve the new formulae. Clearly the new engine does not, in general, produce formulae that are faster to solve, however, when looking at the special case of recursion boolector solves the new formulae faster than the old ones. Apparently the now proper handling of recursions also seems to produce easier to solve formulae, interestingly enough there seems to be a trade off, though, as the other examples take longer to solve.

However, these results should be considered with some caution, as they highly depend on the inner workings of boolector, which is very complicated in itself.

Chapter 5

Conclusion and Outlook

Time to reiterate; we had the symbolic execution engine Christian Edelmayer worked on in order to compare symbolic execution and bounded model checking. It still had problems in dealing with some special cases of recursion, like the recursive implementation of the Ackermann function. In the course of this project we addressed the issue with recursion and ultimately completely changed the schedule of the monster engine.

What actually changed in the course of this project?
The monster engine in Selfie was reworked in such a way that the contexts during the symbolic execution of a program with a recursive procedure call are all merged properly. Furthermore, some assumptions about how the compiler translates source code to machine instructions were relaxed with the new implementation. These improvements were made possible by first storing the call stacks of the individual contexts in order to determine the recursion level of any given context and only merge those which are on the same level. The call stacks are now stored in a global tree structure and the individual contexts only store a pointer which points to the node in that call stack tree that corresponds to their call stack. This way we save memory, since we do not need to store a call stack per context and we enable different call stacks to be compared in constant time. Secondly the schedule of the context was changed completely to be more general. Instead of pre-calculating merge locations and scheduling the contexts such that they would wait at those pre-calculated locations, the context that is furthest behind is executed for one instruction and then it is checked whether it can be merged with any other context. Then the next context to be executed is chosen.

The criterion to be selected is that the context must have the biggest call stack and from all the contexts with that call stack the lowest program counter.

As with many projects at the end there is still a lot more to refine and extend. One aspect one could still optimize in this engine is how long it takes to schedule the next context. As of now the contexts are stored in a linked list, which means finding the context with the biggest call stack and lowest program counter takes linear time in the number of contexts. An idea to improve this would be a nested heap structure, where the inner heap is a min-heap which simply keeps the context with the lowest program counter at the first position. All the contexts in such a min-heap would have the same call stack. And the outer heap would be max-heap, which contains pointers to the min-heaps and always keeps the pointer to the min-heap with the biggest call stack in its first position. This would give us constant access time for the context that needs to be scheduled next. However, this heap structure would have to be reorganized maybe not after every instruction, but very often, which is also a cost that would need to be considered.

Bibliography

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[2] 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.

[3] E. W. Dijkstra. "Recursive Programming". In: *Numer. Math.* 2.1 (Dec. 1960), pp. 312–318. ISSN: 0029-599X. DOI: 10.1007/BF01386232. URL: https://doi.org/10.1007/BF01386232.

[4] Christian Edelmayer. *Hybrid Symbolic Execution and Bounded Model Checking Engine in Selfie*. 2019. URL: https://github.com/cksystemsteaching/selfie/blob/master/theses/bachelor_thesis_edelmayer.pdf.

[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: https://doi.org/10.1145/360248.360252.

[6] 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: Association for Computing Machinery, 2017, pp. 198–213. ISBN: 9781450355308. DOI: 10.1145/3133850.3133857. URL: https://doi.org/10.1145/3133850.3133857.

[7] Christoph M. 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 07/25/2020).

[8] 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: https://doi.org/10.1145/2345156.2254088.

[9] 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.

[10] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75.