Bachelor Thesis

Symbolic Execution with Selfie
Arithmetics

Manuel Widmoser
mwidmoser@cs.uni-salzburg.at

Department of Computer Sciences
University of Salzburg
Austria

August 18, 2018

Advisor
Univ.-Prof. Christoph Kirsch
ck@cs.uni-salzburg.at

Contents

Abstract

Modern symbolic execution engines are lacking in scalability and performance due to computationally expensive *Satisfiability Modulo Theory* (SMT) solvers. Such engines attempt to execute for all possible inputs in order to find vulnerabilities in an elegant way. Within an entirely self-contained system, we developed a symbolic execution engine on machine code level without using a theory solver. This becomes achievable by interval arithmetic and the ability of backtracking code execution. However, there is a trade-off between performance and completeness.

This thesis mainly describes the data structure behind the engine as well as symbolic code execution using interval arithmetic. Despite some reasonable limitations the subset of programs, which can be executed by the engine, keeps sufficiently interesting. In return, soundness and simplicity maintain.

1 Introduction

Software vulnerabilities may be exploited in order to run malicious code or gain access to sensitive data. Therefore it becomes of increasing importance to develop correct software without having any security flaws. This work gives an approach of exhaustive code verification based on symbolic execution.

Chapter 1 concisely describes the used environment, states the problem and gives a brief overview of symbolic execution as well as some concepts related to this work. The skeleton of the engine, thus the essential data structure and the path towards it, is described in the second chapter. Chapter 3 outlines how instructions are executed from an arithmetical point of view. Moreover, how constraints are being prepared, which is crucial for generating satisfying assignments. With a satisfying assignment, also called *witness*, an explored path can be triggered again, *i.e.*, a path including a bug is reproducible. The third chapter further describes overflow handling as well as limitations that restrict the engine. Finally, the last chapter concludes the work, states remaining problems and gives suggestions for further work.

1.1 Symbolic Execution

Symbolic execution is a program analysis technique for testing software against properties that can be violated and is used to reason about software path-by-path [1, 7]. Since it is desirable to develop correct and secure programs (no `NULL`-pointer dereferencing, no division by zero, no security backdoors), symbolic execution became popular in the last decades especially in software security and software verification.

In concrete execution, a single control flow path is executed, *e.g.*, a program-tester assigns specific input values. Whereas in symbolic execution all possible paths are being explored. That becomes achievable by assigning input variables symbolic values rather than concrete values. For each control flow path, a first-order Boolean formula maintains, called *path condition*, which can be finally solved.

Satisfiability may be determined by feeding per path the logical formula into a theory solver [10, 3]. Such an SMT solver provides in the best case either (1) a range of all possible input values, that allows reproducing the path, or (2) declares the formula unsatisfiable. If a formula is unsatisfiable, a particular path is unreachable for any valid input. In the worst case, the theory solver never converges and runs forever.

The symbolic execution engine KLEE found fatal errors in heavily tested code, for instance in *GNU Coreutils*, including bugs that had escaped detection for 15 years [4].

1.2 Selfie

Selfie[1] is a self-contained 64-bit 10-KLOC implementation of a self-compiling compiler (`starc`), a self-executing emulator (`mipster`), as well as a self-hosting hypervisor (`hypster`) written in C*, a minimal but Turing-complete subset of C. Meanwhile, Selfie supports a 64-bit instruction set called `RISC-U`, which is a tiny easy-to-teach subset of `RISC-V`. Selfie was originally developed for educational purposes and even can compile, execute and virtualize itself any number of times and can be arbitrarily stacked [8, 9]. Today, students and researchers are working with Selfie for different scientific aims. Our symbolic execution engine is implemented within Selfie.

1.3 Problem Definition

State-of-the-art symbolic execution engines are based on computationally expensive SMT solvers that may not perform well on large programs. There is, however, a trade-off between performance and completeness. Restricting completeness such that a solver works only on a subset of all possible programs may nevertheless allow us to make the solver faster and scale to larger programs. The challenge is to be able to detect if a solver is still complete during symbolic execution of a given program. Moreover, the subset of programs for which a solver may be complete during symbolic execution needs to be sufficiently interesting. For us, this is the case if non-trivial parts of our selfie system are part of that subset.

1.4 Capabilities

Our symbolic execution engine is far away from being complete. In general, it attempts to execute all possible paths that may be achievable depending on a symbolic input value. Further, the engine determines for every path whether it is satisfiable or not and reports possible input values. Such values allow triggering the path once again.

Nevertheless, an engine without using a theory solver brings restrictions. Reasoning about single bytes within a symbolic buffer, e.g., characters within strings, is currently not working properly. A heavy limitation is that for almost every arithmetic operation at most one operand is allowed to be symbolic. This might be due to our backward propagation approach, which is described in [11]. Therefore the engine is not able to constrain more than a single symbolic value simultaneously without losing information. Moreover, overflow handling for the modulo operation is not supported.

[1] http://selfie.cs.uni-salzburg.at

1.5 Intervals

Instead of applying costly theories, *e.g.*, bit-blasting [10], an engine based on interval arithmetic leads to significantly better runtime behavior. In our design, a single path can be solved in linear time in the number of instructions. However, this is only true for a designated subset of programs.

Instead of passing logical formulas along all paths and solving them after all, our idea was to represent a symbolic value as an interval. When a symbolic value becomes initialized, the interval starts from 0 to the highest possible value. Whereas the highest value is in our case $2^{64} - 1$ (abbreviated to `MAX`) since Selfie is a 64-bit implementation allowing unsigned integer arithmetic only. Furthermore, unsigned integer arithmetic significantly simplifies reasoning about correctness.

In this work, the following notation for intervals is used and an unsigned interval defined as follows:

$$[a, b] \mid 0 \leq a \leq b < 2^{64}$$

Typically, symbolic execution engines are constraining symbolic values when new paths originate, meaning that the number of possible satisfying assignments becomes incrementally smaller. Such assignments can be ultimately determined by solving the *path condition*. The concept of constraining values is applicable on intervals as well and is described further in chapter 3.3.

1.6 Workflow

The algorithm of the engine mainly consists of three parts: (1) forward execution, (2) backward constraining and (3) path exploring. During forward execution, a single path is executed and starts from where the symbolic value origins. If branching occurs, *e.g.*, through an `if` condition, the forward execution algorithm pre-computes all possible constraints and stores them on the used data structure, the `trace`. Such constraints, for instance, might look as follows for `if (x < 10)` under the assumption that `x` is symbolic: $[0, 9]$ would be the `true` constraint and $[10, \texttt{MAX}]$ the `false` constraint. Generally, these constraints are necessary in order to generate satisfying assignments. The forward pass stops whenever the currently executed path hits an `exit` system call or runs out of memory.

Thereafter, the algorithm switches to backward constraining, executes all instructions inversely and applies the pre-computed constraints. When the origin of the symbolic value is reached, the backward pass is done and finally provides all possible witnesses that can trigger the processed path. If there is no satisfying assignment remaining, the engine claims the path as unsatisfiable.

Subsequently, path exploring begins and tries to find the nearest still unexplored path by doing it in a depth-first-search like way. This is attainable by looking for remaining constraints. Simultaneously, the algorithm undoes registers and memory values until that point. Thus regenerates an earlier machine state. Path exploring is either

done when reaching the very first instruction (the beginning of the program), thus no path is remaining, or by finding a constraint at any branch instruction. Then another path is chosen, *e.g.*, through switching from the `true` branch to the `false` branch, and again the forward pass continues.

Figure 1 visualizes the algorithm of the engine and its workflow.

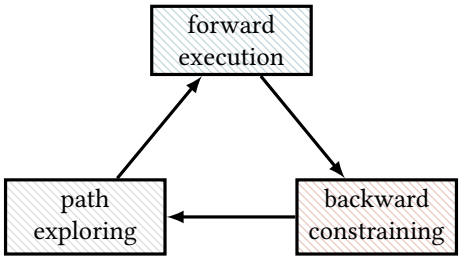

Figure 1: Workflow of the engine

Before forward execution comes in again, the algorithm is located at a branch position if another unexplored path is remaining. Hence the forward pass does not begin from where the symbolic value origins, but rather from the branch location itself. This prevents from redundant code execution and is described in greater detail in chapter 2.2.5. The whole process repeats until all paths have been explored entirely, except the engine runs out of memory priorly.

1.7 Contributions

The implementation of this project was a collaborative work between *Simon Bauer* [2], *Sara Seidl* [11], and *Manuel Widmoser*. Each of us has implemented parts of the engine independently. For rewriting system calls, memory bootstrapping and exhaustively testing the engine, *Simon Bauer* was responsible. *Sara Seidl* implemented the most recent data structure, memory synchronization as well as the foundation for reasoning and generating satisfying assignments. Interval arithmetic, pre-computing constraints, and overflow handling were done by myself.

However, most features, as well as many other details, were discussed and planned as a group priorly. Moreover, I want especially acknowledge our advisor *Christoph Kirsch* for providing us with input and tons of word of advice.

2 Data Structures

Designing a smart and versatile data structure was crucial since it is the scaffolding of the whole engine. While developing ideas for an ideal data structure some criteria established gradually:

- No dynamic memory allocation,
- constant time access,
- backtracking of code execution,
- appropriate representation of intervals.

In symbolic execution path explosion is an unavoidable problem. At every branch of the program, new paths may emerge, *e.g.*, due to `if` or `while` conditions. Therefore the total number of paths can rapidly become exponential in the number of branches [1]. Consider the code of listing 1 under the assumption that x is a symbolic value.

```
1  uint64_t i = 0;
2
3  while (i < 10) {
4    if (x < 5)
5      doSomething();
6
7    i = i + 1;
8  }
```

Listing 1: Simple path explosion example

In each iteration two additional paths emerge: $x = [0, 4]$ and $x = [5, \texttt{MAX}]$, which results in $2^{10} = 1024$ different paths for merely ten iterations. That is the case since constraints are applied not until during back-propagation in our engine (see chapter 3.3). Moreover, dynamic memory allocation would undesirably blow up memory usage. For instance, when the engine would allocate memory for each path that emerges at runtime separately.

The capability of backtracking code execution becomes of increasing importance when a witness for a specific path should be found. This approach is even quite useful to choose the next path by directly following the chain back, which the data structure creates for us, to the latest non-processed branch instead of starting from the beginning again.

2.1 First Approaches

The very first idea that we pursued was constructing expression trees to represent symbolic values. Expression trees are a common technique and are often used in symbolic execution engines [4, 7]. Such a tree can then be treated similarly like a path condition

and passed to a theory solver after all. They grow during execution and can be deallocated subsequently. However, there is no `free` in Selfie, meaning that deallocating memory from the heap is hardly possible, which was a major reason for discarding this approach.

Backtracking was already at the beginning a hot topic since we were searching for a solution where code execution can be recorded. Another approach for a data structure design was to organize the trace generation directly on the stack during execution. Despite enumerating all instructions, this approach led to a shuffled trace of instructions. In order to facilitate backtracking, we had to reorder the gained trace, *e.g.*, with a sorting algorithm. For such an engine, however, is sorting computationally expensive and therefore no option.

The latest approach was quite similar to the final data structure design. At this point, the decision to represent symbolic values as intervals was already made and we introduced tiny structures. These structures were called *containers* and had the ability to store the lower and upper bound of an interval and even more essential information. Registers as well as entries in memory dealt with pointers to these containers instead of actual values. Containers were only allocated on demand and therefore this approach was quite memory space efficient. Code execution seemed to work fairly uncomplicated, however, the ability of backtracking was still pending.

Ultimately, the final data structure design interconnects all desired features.

2.2 Trace

The used data structure, which we call `trace`, since it is a trace of instructions, can be seen as a two-dimensional array and looks as follows:

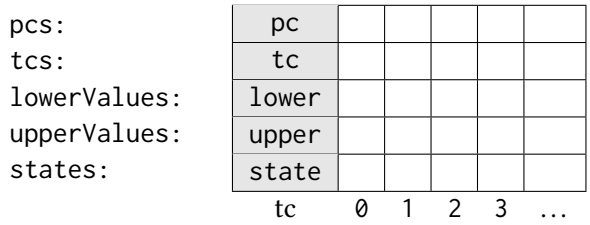

Figure 2: Organization of the trace

The trace is more or less a versioning system for registers and memory values. Each entry in the trace gets its own unique tag and is called the *trace counter* (`tc`). The `tc` allows constant time access by accessing the trace like an array with a specific index. One can find five fields for each trace entry:

- **`pc`:** The program counter at which instruction the current trace entry was generated.
- **`tc`:** A trace counter which points to the previous entry before updating the register or memory value.

- **`lower/upper`:** Both together represent an interval, the lower and upper bound, respectively.
- **`state`:** An integer flag which represents a specific state of an interval: `concrete`, `symbolic`, `constrained` or `unsatisfiable`.

It is important to understand that registers and memory are solely dealing with these version tags (`tc`). Such a `tc` ultimately represents an entry in the trace, thus represents an interval and further information. Moreover, if one accesses a register, *e.g.*, by `*(registers + rd)`, which would reveal the content of the destination register `rd`, in the end one would get a `tc` that points to a trace entry.

This holds for memory as well. Accessing virtual memory, thus loading a 64-bit word from a virtual address, returns a `tc` too. Again, memory and registers do not know anything about intervals, merely about `tc`s.

2.2.1 Fields

The following four paragraphs describe the individual fields of a single trace entry in more detail.

2.2.1.1 Program Counter

The program counter (`pc`) field is not only responsible for representing the machine state including the involved instruction. Sometimes it is necessary to remember additional information for a particular instruction. Hence, further entries on the trace can be created. For instance, when executing a division operation (`divu`), two entries on the trace are placed rather than one: The actual result, thus the quotient, and additionally the remainder, which is important to regain a symbolic interval correctly during the backward pass. More details are given in chapter 3.2. The crux is that the `pc` field may be exploited to check whether there are even more entries for the same machine state.

2.2.1.2 Trace Counter

To allow backtracking of code execution the trace counter (`tc`) field is indispensable. Usually, when a register or memory value becomes overwritten, previously processed information get lost. For exploring a single control flow path, as in concrete execution, this does not matter, since there is no need for backtracking code execution. Nevertheless, to generate satisfying assignments in a relatively simple manner, we pursued the approach of executing the code backward. In order to be able to restore values properly, it is crucial to know what a specific register or memory value carried along before an instruction has been executed.

Assume that a store instruction, an instruction which stores from a register to a stated memory location, overwrites an already existing memory value during forward execution. When going backward, the store instruction needs to be executed inversely and the goal is to retrieve the old value again. The inverse store instruction is a load

instruction, which loads from a stated memory location to a given register. However, how do we know what interval was at the stated memory location before executing the store instruction? Before overwriting a register or memory value ultimately, the trace counter of the preceding value is stored in the newly generated `tc` field, which basically points to the previous trace entry and in turn, represents an interval plus some additional information. This step appears in every instruction where updating a register or memory value occurs. Hence, during backward execution, it is quite easy to regain such values by accessing the trace at the position at which the trace counter of the `tc` field points to.

2.2.1.3 Interval Boundaries

An interval is stored as a tuple in the trace, namely the lower and the upper bound. For instance, figure 3 denotes a symbolic interval $[0, 255]$. In contrast, to represent a singleton interval, *i.e.*, a concrete value, both boundaries are naturally equal.

`lower`	0
`upper`	255

Figure 3: Representation of an interval by the trace

2.2.1.4 State Flags

To maintain certain properties of intervals, we introduced the `state` flag. A `concrete` flag obviously claims a concrete value. Why is such a flag necessary since a singleton interval already represents a concrete value? This is only partially true, imagine a symbolic value and a concrete value that lies within the symbolic interval. Furthermore, compare them by using a logical equality operator, *e.g.*, `[0,255] == [10,10]`. The outcome for the true branch would yield $[10, 10]$, thus a singleton interval, however, it is still symbolic. During backward execution, *i.e.*, when propagating the constraints, the engine must know which interval is symbolic in order to constrain it properly. Would not there be such a flag, both intervals are detected as concrete, hence no constraining occurs which finally leads to a wrong generation of satisfying assignments and in the end, the engine would become unsound.

A symbolic interval obtains the `symbolic` flag when it becomes initialized. If an `SLTU` (*Set on Less Than Unsigned*) instruction computes constraints for a symbolic interval, as we will see in chapter 3.3, the resulting interval is denoted as `constrained`. Whenever backward execution decides unsatisfiability for a certain path, as described in [11], the interval is stated as `unsatisfiable`.

These flags are strictly totally ordered: `concrete` (1) < `symbolic` (2) < `constrained` (3) < `unsatisfiable` (4). Meaning that a symbolic stated interval can never gain or regain the `concrete` flag, et cetera. This preserves among others that an already unsatisfiable path could become satisfiable again, which coincidentally could appear during

back-propagation. Instructions which are generating new trace entries either inherit or update such a flag quite easily due to the property of total ordering.

2.2.2 Implementation

Our trace, thus the two-dimensional array with size $5 \times$ `maxTraceLength`, is implemented in C* by instantiating five pointers of type `uint64_t*`. Figure 2 exhibits the five single arrays on the left side. Each of them gets preallocated with `maxTraceLength` $* 8$, whereas 8 (bytes) the size of an unsigned 64-bit integer represents. `maxTraceLength` is a predefined constant which limits the memory consumption.

Accessing an entry with a given trace counter can be achieved by dereferencing such a pointer. For instance, `*(lowerValues + 42)` accesses the lower interval bound of entry `tc 42`. Every time an instruction tries to generate a new trace entry, a safety check ensures whether the trace exceeds. If so, the engine needs to terminate due to lacking memory.

2.2.3 Bootstrapping

Before starting with actual execution some bootstrapping issues need to be resolved. Whenever Selfie begins emulating code it firstly initializes a given amount of memory, creates a context, *i.e.*, the memory layout, and loads the binary. Essentially, it maps the code and data segment into memory, sets the program break properly, et cetera. The data segment basically contains global variables and strings. However, the concept of using memory changed since we are now dealing with trace counters. Trace counters are pointing to the trace and ultimately expose the memory value, rather than accessing memory values directly. Hence, it is necessary to scan through the entire data segment, push global variables and strings on the trace, *i.e.*, create new trace entries, and store the corresponding `tc`s at the respective memory location. Otherwise, when later on in code execution one would try to access a global variable or a string from the data segment, an illegal trace access would occur. That is comparable to an index out of bounds error and most likely a segmentation fault would arise.

```
1  while (codeLength < GP) {
2    GP = GP - REGISTERSIZE;
3
4    if (isValidVirtualAddress(GP + entryPoint)) {
5      if (isVirtualAddressMapped(table, GP + entryPoint)) {
6        setConcrete(loadVirtualMemory(table, GP + entryPoint));
7        setStateFlag(CONCRETE, tc);
8
9        storeVirtualMemory(table, GP + entryPoint, tc);
10       incrementTc();
11     }
12   }
13 }
```

Listing 2: Data segment preparation for trace usage

Listing 2 shows how the data segment is prepared before the symbolic execution engine starts running code. `entryPoint` is the actual entry point of the code segment in virtual address space and is encoded in the *Executable and Linkable Format* (ELF) header [12], which the compiler creates. After a binary is loaded, the ELF header can be extracted effortlessly to find some required information. The `codeLength` variable denotes the length of the binary without global variables and strings and can also be decoded from the ELF header. This is exactly the address where the data segment begins, thus the border between code segment and data segment as shown in figure 4.

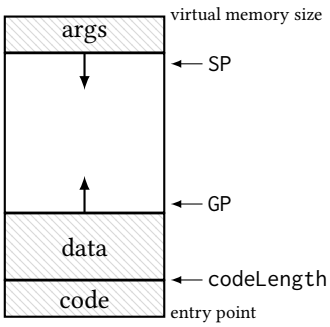

Figure 4: Memory layout that Selfie generates

At this point, the global pointer (`GP`) is already set and is located directly after the data segment. This makes scanning the entire data segment fairly easy. We scan double word by double word (`REGISTERSIZE` is equal to 8) and check whether the virtual address is valid and already mapped. Subsequently, up to 8 bytes are loaded from memory and stored on the trace as a new entry. `setConcrete` simply sets the lower and upper bound of the interval equally. Finally, the associated trace counter is stored at that memory location and gets increased afterward. Therefore, the `tc` always points to the next available trace entry.

There are similar issues for program arguments and registers, in particular, system registers like stack pointer and global pointer. Program arguments are stored at the topmost location as one can see in figure 4. Hence, pushing the arguments on the trace works just like before, except the scanning area is now from the address at which the stack pointer is located until the highest possible virtual address.

Values of system registers (stack pointer, global pointer, frame pointer, return address register) are stored on the trace and replaced by `tc`s as well.

2.2.4 Trace Example

In order to fully understand the workflow of the trace and how to access it, let us consider the code example in listing 3. The example shows how a symbolic value is actually initialized, which is ultimately done by a `read` system call and is described in more detail in [2]. To be more precise, a symbolic value is indeed a symbolic buffer and can even be larger than 64-bit. It depends on how much memory is allocated for the

symbolic value and how many bytes of them are assigned to be symbolic. This is why `x` is of type pointer. In the example, the symbolic buffer is initialized with 8 bytes at line 4, however, only a single byte is defined to be symbolic (the third parameter of the `read` call). Thus the variable `x` represents an one byte symbolic interval, in fact $[0, 255]$.

This setup was chosen to simplify the example. Usually, 8 bytes are assigned to be symbolic, which leads to the typical $[0, \texttt{MAX}]$ interval and represents all possible values of a 64-bit variable.

```
1  uint64_t* x;
2
3  uint64_t main() {
4    x = malloc(8);
5    read(0, x, 1);
6
7    *x = *x + 5;
8
9    return 0;
10 }
```

Listing 3: Code example of adding a concrete value to a symbolic interval

The most interesting part in the example is how a symbolic buffer can be mutated with respect to the trace. Line 7 provides loading a symbolic value, adding a concrete value to it and storing it finally back to memory.

First of all it is necessary to load `x` into an available register, *e.g.*, temporary register `$t0`. Since it is globally declared, `x` must be in the data segment, thus the address of `x` is the address of the global pointer minus some offset.

As already mentioned in chapter 2.2.3 are addresses that are stored in system registers, *e.g.*, global pointer (`$gp`), put on the trace and replaced by trace counters. In our example the address of the global pointer is already on the trace: Assume `0x0F` is the actual address and the register `$gp` itself stores the trace counter 1. Figure 5 illustrates the corresponding view of the trace. The `tc` field is empty since there is no predecessor for the global pointer register.

Possibly someone might have noticed that starting with trace counter 1 is not entirely correct. This is due to the fact that further information, for instance, addresses of other system registers or information about the first assignment of `x` (line 4), must be provided by the trace as well. For keeping the example simple, only necessary information is pushed on the trace though.

Furthermore, we reveal the trace counter stored in the global pointer register, subtract 8 as offset, since there is just a single global variable, and load the memory entry at which the resulting address points to into register `$t0`. However, this entry is not `x` itself, rather again a trace counter which represents `x` ultimately. In figure 5 therefore `tc 2` represents the already existing memory entry and `tc 4` the currently updated register entry. More precisely, since `x` is of type pointer, the trace entry of `x` represents another address and needs to be dereferenced, *i.e.*, the address needs to be loaded once again. To keep the address, which is required to store the result finally back to memory, another available register, *e.g.*, `$t1`, is used to dereference `x`. Hence, `*x`, the actual

symbolic value, is represented by the trace counter 3 and a new trace entry associated to `tc 5` is stored into register `$t1`. The temporary register `$t1` simply takes the interval from the memory entry of `*x`.

pc				LD	LD	ADDI	ADD	SD
tc	-	-	-	-	-	-	5	3
lower	0x0F	0x17	0	0x17	0	5	5	5
upper	0x0F	0x17	255	0x17	255	5	260	260
state	conc	conc	sym	conc	sym	conc	sym	sym
tc	1	2	3	4	5	6	7	8
	↑	↑	↑	↑	↑	↑	↑	↑
	$gp	0x07	0x17	$t0	$t1	$t2	$t1	0x17

Figure 5: Trace view of adding a concrete value to a symbolic interval

The next step is creating a new trace entry for the concrete value 5. Another available temporary register `$t2` stores the trace counter of this trace entry, which is 6 in our example.

An addition of the registers `$t1` and `$t2` is performed subsequently. Thus the lower and upper bounds of the entries represented by the trace counters, which are carried by the stated registers, are added together. The result is finally stored as a new trace entry associated to `tc 7`. Registers `$t1` and `$t2` are no longer needed and therefore `$t1` can be overwritten with the resulting trace counter. Notice that trace entry 7 stores in the `tc` field the trace counter that register `$t1` carried before. This allows backtracking of code execution.

Finally, storing the result back to memory is desired, which is the actual assignment in listing 3. Fortunately, the address is still represented by the trace counter stored in register `$t0`. We update the memory entry at address `0x17` by a new trace entry that gives the final result [5, 260].

The program counter field (pc) usually stores the program counter of the instruction which was responsible for creating the corresponding trace entry. However, in our example, the meaning of an empty field is that the associated instruction was already executed before line 7 in listing 3. Thus the trace entry already existed.

For the sake of completeness, listing 4 below shows how `*x = *x + 5;` is processed by the machine as a sequence of RISC-V instructions [13]. The association between the assembly code and the trace might be interesting.

```
1  LD   $t0 -8($gp)
2  LD   $t1 0($t0)
3  ADDI $t2 $zero 5
4  ADD  $t1 $t1 $t2
5  SD   $t1 0($t0)
```

Listing 4: Assembly code of adding a concrete value to a symbolic interval

The changed semantics of all RISC-V instructions for symbolic values are further described in chapter 3.2.

2.2.5 Successive Trace Generation

Chapter 1.6 outlines the three main parts: (1) forward execution, (2) backward constraining and (3) path exploring, thus the workflow of the engine. The interconnection between the actual workflow and the trace generation is visualized in figure 6 and 7.

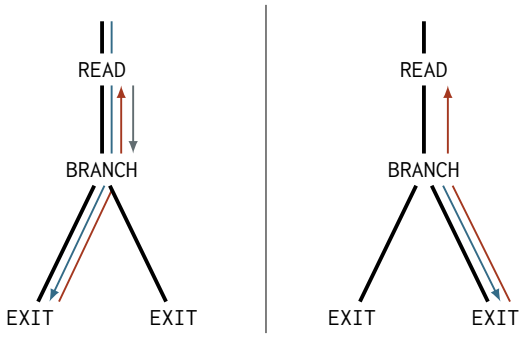

Figure 6: Logical workflow of the engine

During forward execution the trace is generated successively as shown in the example above. Backward constraining logically executes the instructions inversely, however, pushes the trace entries after the *execution break*. The execution break depicts where forward execution is done, meaning at which position the currently executed path of the program stopped, *e.g.*, by an `exit` system call.

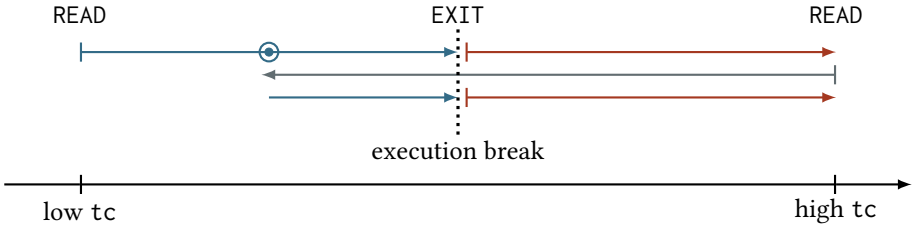

Figure 7: Trace generation strategy of the engine

Despite the algorithm goes back to the point where the symbolic value originates, the amount of trace entries redoubles. It is necessary to keep that information in order to be able to fully reconstruct the machine state. That is crucial for exploring further paths.

While looking for another path, registers and memory values receive their previous values successively. The machine state is just like before the latest branch occurred if another unexplored path exists. Otherwise, all paths are explored and the engine successfully stops. New trace entries are generated by replacing redundant entries, *i.e.*, forward execution resumes from the latest branch position (denoted by the circle in figure 7), and a new path becomes explored.

3 Arithmetics

In the introduction, intervals and their notation were already defined. This chapter gives greater details, how intervals behave during execution, how they are implemented in instructions as well as some corner cases, *e.g.*, overflows.

The interval space in our 64-bit system consists of exactly 2^{64} values. Moreover, this means also that a single interval contains at least one value and at most 2^{64} values, which is then called a full interval. In our engine, an empty interval is undefined and therefore needs no longer to be considered. An interval containing a single value is called a singleton interval. Concrete values are always represented as singleton intervals, *e.g.*, $[5, 5]$, which represents an unambiguous value. Symbolic values are therefore non-singleton intervals.

A full interval is mostly denoted as $[0, \texttt{MAX}]$, however, a wrapped-around interval can also be a full interval. A wrapped-around interval is an interval where an over- or underflow occurred. For instance, $[5, 4]$ is wrapped-around and equivalent to $[0, \texttt{MAX}]$ since it contains the whole range as well and therefore all possible values from 0 to MAX. Wrapped-around intervals are described further in chapter 3.4.

An interval can be simply visualized as in figure 8. For representing a modular interval a circle suits well due to wrap-around semantics.

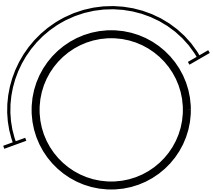

Figure 8: Representation of a modular interval

The system executes arithmetic operations by using *unsigned* integer arithmetic with wrap-around semantics. Contrarily, in signed integer arithmetic, the most significant bit (leftmost bit) denotes whether the value is positive or negative. Hence, possible values would be in a range starting at -2^{64-1} up to $2^{64-1} - 1$. However, unsigned integer arithmetic can only represent non-negative values, thus a range starting at 0 up to $2^{64} - 1$. The highest possible value (MAX) is therefore $2^{64} - 1$.

Wrap-around semantics means that numeric values can over- or underflow. Such an integer overflow occurs when an arithmetic operation attempts to create a value which is outside of the range that can be represented. In mathematical terms, wrap-around means executing an arithmetic operation modulo $2^{\#\text{bits}}$.

3.1 Cardinalities of Intervals

The cardinality of an interval [5], which is the number of distinct values within the interval, is defined as follows:

$$\#([a, b]) = b - a + 1$$

Notice the full interval $[0, \texttt{MAX}]$ has cardinality 0, since $\texttt{MAX} - 0 + 1 = 2^{64}$ and $2^{64} = 0$ due to wrap-around semantics. Whereas a singleton interval, *e.g.*, $[5, 5]$, always has cardinality 1, obviously, since the interval only contains the single value 5.

For detecting whether an arithmetic operation on an interval results in an invalid range, cardinalities are indispensable. For instance, consider the interval $[0, 2^{63}]$ multiplied by 2, which would yield $[0, 2^{64}] = [0, 0]$ and is certainly wrong. Such results would lead to unsoundness, meaning that too few or too many values are considered and therefore the reasoning would be inaccurate. An interval cannot become larger than a full interval thus the correct outcome would be $[0, \texttt{MAX}]$.

Each operation has its own cardinality-check, some of them are based on [5] but slightly altered to fit the systems design. They are further described in chapter 3.2.

3.2 Instructions

In order to guarantee a proper cooperation between the data structure and actual code execution, all 14 `RISC-V` instructions within Selfie have been reimplemented. The behavior of some instructions remains the same except for that they have to access the trace. Reading or writing actual values directly from registers or memory addresses is not sufficient anymore. Rather points the trace counter to an entry on the trace which reveals an interval.

However, for arithmetic instructions ($+$, $-$, $*$, $/$, $\%$) the semantics considerably changed. Since unsigned modular interval arithmetic is defined differently than for standard unsigned integer arithmetic.

`SLTU` (*Set on Less Than Unsigned*) is particularly interesting which is responsible for pre-computing constraints and overflow handling. `SLTU` is described in chapter 3.3 and 3.4, respectively.

3.2.1 Memory Instructions

Memory instructions are required for accessing virtual memory. In particular, there are two available: *load* for reading from memory and *store* for writing to memory.

3.2.1.1 Load

A load instruction loads a 64-bit value from a given virtual address, carried by a source register (`$rs1`), plus some offset into a destination register (`$rd`). Since our system is double word aligned, the offset, as well as the address, has to be a multiple of 8.

Below one can find the RISC-V instruction of *load double word* (LD).

```
LD $rd offset($rs1)
```

The given virtual address may not be a symbolic value since accessing memory must be unambiguous. This is kind of natural, addresses are managed by the system internally and a piece of software should not be able to manipulate the address space either way. Therefore it is not allowed, for instance, to use a symbolic value as an offset in order to access a buffer as shown in listing 5. Hence, first of all, it is crucial to verify whether the tc carried by the source register points to a trace entry that is stated as concrete. Otherwise, the engine raises an exception.

```
1  uint64_t* x;
2  uint64_t* buffer;
3
4  x = malloc(8);
5  read(0, x, 8);        // x = [0,MAX]
6
7  buffer = malloc(16);  // arbitrary buffer
8
9  *(buffer + *x) = 5;   // illegal access
```

Listing 5: Illegal memory access caused by a symbolic value

Additionally needs to be checked whether the address is valid and already mapped. However, this is done as before and finally, the memory value can be safely loaded. As a reminder, the loaded value is as expected another trace counter. A new trace entry, which inherits the lower and upper interval bound as well as the state from the obtained tc, is created. The associated trace counter is stored into the destination register. Moreover, the already overwritten tc, previously carried by the destination register, is saved in the trace entry as well. Similarly as described in the example of chapter 2.2.4.

3.2.1.2 Store

Contrary, a store instruction writes 64-bit to a given memory location and can be considered as the inverse LD instruction. Equally to load, the source register $rs1 holds a virtual address and an *immediate* value specifics the offset. Another source register ($rs2) contains the value that will be stored in memory.

The *store double word* (SD) RISC-V instruction is described as follows:

```
SD $rs2 offset($rs1)
```

In order to run properly on our symbolic execution engine, it is crucial to prove again whether the given address is valid, already mapped and especially a concrete value. Naturally, values stored in registers and memory are trace counters which represent a trace entry.

3.2.2 Control Flow Instructions

Control flow is the order in which individual statements or instructions are executed. At the level of machine code, control flow instructions are conditional or unconditional *branch* instructions. However, all other instructions affect the control flow as well by altering the program counter (*trivial control flow*), but are not denoted as control flow instructions.

This section is kept quite briefly since those instructions hardly differ from the usual implementation, except for taking the data structure into consideration. Thus previous register values are tracked by the trace and register values themselves are trace counters as usual.

In `JAL` and `BEQ`, addresses are *immediate* values anyway, therefore no attention whether a value is concrete or symbolic must be payed. This is called *pc-relative addressing* and increases or decreases the program counter by a given offset (address).

The implemented `RISC-V` control flow instructions are described below.

```
JAL   $rd   imm
JALR  $rd   $rs1  imm
BEQ   $rs1  $rs2  imm
```

3.2.2.1 Jump and Link

`JAL` is used to call procedures, since saving the link allows the procedure to return to the point of the call. Meaning that the return address, which is the address of the next instruction, is stored in destination register `$rd`.

3.2.2.2 Jump and Link Register

Instead of pc-relative addressing uses `JALR` *register-relative addressing*. Hence, the address comes from the source register `$rs1` plus some offset. This allows greater jumps since *immediate* values can store 12 bits only. The link, *i.e.*, the next instruction, is stored in the destination register as in `JAL`.

3.2.2.3 Branch on Equal

`BEQ` is used for conditional jumping, *i.e.*, in cases of `if` or `while`. If the value stored in source register one (`$rs1`) matches with the value in source register two (`$rs2`), the jump is executed. Due to the code generation of `starc`, Selfie's compiler, register `$rs2` is always the zero register. Therefore the value in `$rs1` is compared against zero. In our case, we check whether the lower interval bound of the trace entry, represented by `$rs1`, is equal to zero.

Typically, an `if` or `while` statement ultimately includes a relational operator such as <, <=, ==, !=, > or >=. The result of the condition is therefore either true or false, thus $[1, 1]$ or $[0, 0]$. Hence, the lower bound is sufficient.

3.2.3 Arithmetic Instructions

Arithmetic operations, such as +, −, ∗, / and %, are processed by arithmetic instructions, which can be found below. These instructions using unsigned integer arithmetic with wrap-around semantics as already mentioned earlier.

```
ADD   $rd  $rs1  $rs2
ADDI  $rd  $rs1  imm
SUB   $rd  $rs1  $rs2
MUL   $rd  $rs1  $rs2
DIVU  $rd  $rs1  $rs2
REMU  $rd  $rs1  $rs2
```

Arithmetic instructions are all operating the same way. The value in source register `$rs1` is the left-hand side of the operator, the value in `$rs2` the right-hand side. After executing the operation, the result is stored in the destination register `$rd`. Add immediate (`ADDI`) forms an exception, the right-hand side is a 12-bit immediate value rather than a value stored in a register.

3.2.3.1 Addition

Adding two intervals is quite straightforward whether symbolic or not. However, there may raise a problem if the cardinality of both intervals together exceeds the amount of the interval space, meaning that a wrap-around occurs which leads to a wrong result. Figure 9 makes this more clearly.

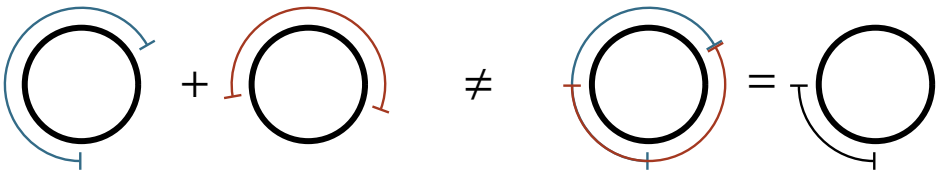

Figure 9: Adding two intervals without cardinality check leads to a wrong result

In this case, the result would definitely be a too small subset. To avoid such an issue, a *cardinality check* for each arithmetic operation is necessary in order to do not overshoot the interval space.

One has always to keep in mind, that an interval represents not only a lower and an upper bound rather than a whole *set* of distinct numbers. That means that an arithmetic operation such as addition applied on two sets is nothing more than adding each value of set one to each value of set two. Since our interval is defined as monotonically increasing and only holds distinct values, the duplicates of the resulting set need to be removed. The minimum value within the set denotes the lower interval bound, the maximum value the upper bound. Sounds computationally expensive, however, it is much easier, this just demonstrated the idea behind.

Addition including the cardinality check is defined as follows:

$$[a, b] + [c, d] = \begin{cases} [a + c, b + d] & \text{if } \#([a, b] + [c, d]) \leq 2^{64} \\ [0, \texttt{MAX}] & \text{otherwise} \end{cases}$$

If the cardinality is greater than 2^{64}, the full interval is the only correct result, since all possible values must appear in the set. The appropriate version of the addition example in figure 9 shows figure 10 correctly.

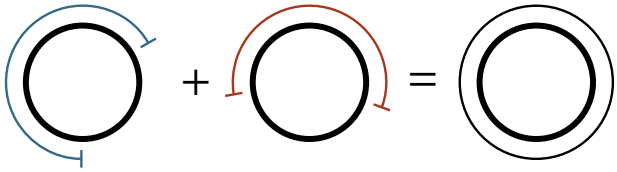

Figure 10: Adding two intervals with cardinality check

Addition does not necessarily enlarge an interval as shown before. For instance, consider a symbolic interval $[0, 255]$ and a singleton interval, *i.e.*, a concrete value, $[5, 5]$. When adding them together, the resulting interval holds the same number of distinct values. Hence, the cardinality of $[0, 255]$ is 256 and the cardinality of $[5, 260]$ is 256 as well. Thus the resulting interval has not become larger, but all values within the interval have been shifted.

A similar example shows figure 11 in order to visualize the *shifting property* of the addition operation. However, the second interval is not a singleton in this case.

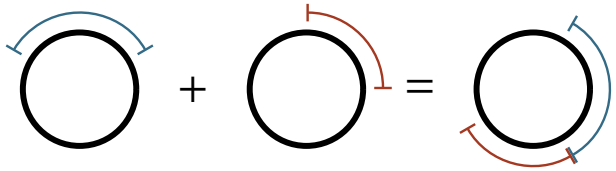

Figure 11: The shifting property of addition

To avoid confusion, figure 9 is lacking the shifting property deliberately.

3.2.3.2 Subtraction

Subtraction can be derived from the properties of how we defined the interval as well as from the interval addition definition:

$$[a, b] - [c, d] \;\Leftrightarrow\; [a, b] + (-[c, d])$$

Due to the monotonically increasing property of our interval definition, we know that $c \leq d$ and therefore holds $-d \leq -c$, hence:

$$[a, b] + (-[c, d]) \;\Leftrightarrow\; [a, b] + [-d, -c] \;\Leftrightarrow\; [a - d, b - c]$$

From that follows that subtraction is quite similar to addition. Therefore the cardinality check remains the same as for addition [5].

The definition of subtraction is defined as follows:

$$[a, b] - [c, d] = \begin{cases} [a - d, b - c] & \text{if } \#([a, b] + [c, d]) \leq 2^{64} \\ [0, \texttt{MAX}] & \text{otherwise} \end{cases}$$

Importantly, subtraction in interval arithmetic is not the inverse operation of addition as it is in standard arithmetic [6]. It is rather comparable to addition, the size of the resulting set can either be larger or equal but never smaller as one would expect from subtraction. That means that the cardinality for $[a, b] - [c, d]$ is exactly the same as for $[a, b] + [c, d]$. The only difference is that the shifting property behaves inversely compared to the shifting property of addition.

3.2.3.3 Multiplication

One of the goals was that soundness for a subset of all programs, executed by our symbolic execution engine, maintains. Therefore, computing intervals must be either explicitly correct or at least be recognizable by the engine that after an operation the subset will be left. When the latter occurs, the engine is not able to reason about the program properly anymore and stops executing.

Computing the cardinality check is therefore particularly important. For that reason, we had to restrict multiplication in a way that one factor has to be a concrete power of two when a symbolic value is involved. To be able to understand this decision, let us consider the cardinality check of addition again in more detail:

$$\#([a, b] + [c, d]) \leq 2^{64}$$

The problem is that the left-hand side can easily overflow due to wrap-around semantics. Thus it must be guaranteed that whether the left-hand side nor the right-hand side can wrap around. Even 2^{64} overflows and leads to 0 and nothing except 0 itself is less or equal than 0. However, after inserting from the cardinality definition (see chapter 3.1) and some rearrangement we get:

$$b + d - (a + c) + 1 \leq 2^{64} \;\Leftrightarrow\; b - a \leq 2^{64} - 1 - (d - c)$$

The left-hand side is dependent from the left addend, hence the highest possible value for $b - a$ is MAX, the lowest 0. On the right-hand side, the -1 guarantees that no overflow can occur.

In order to do not overshoot the resulting interval, similarly to addition and subtraction, it needs to be checked for multiplication whether:

$$\#([a,b]\cdot[c,d]) \leq 2^{64} \;\Leftrightarrow\; bd - ac \leq 2^{64} - 1 \;\Leftrightarrow\; bd - ac < 2^{64}$$

In general, guaranteeing precisely that no wrap-around may occur seems to be infeasible for all possible values, at least for integers. This leads to the first restriction, that one factor has to be concrete. Substitute c for d:

$$bc - ac < 2^{64} \;\Leftrightarrow\; b - a < \frac{2^{64}}{c}$$

Dividing by c is only accurate with a zero remainder, *i.e.*, c must be a power of two, which leads to our second restriction. Substitute 2^{ω} for c and we get:

$$b - a < 2^{64-\omega}$$

The left-hand side cannot wrap around and the right-hand side either, except $\omega = 0$, however, that can easily be caught.

Therefore multiplication is defined as follows:

$$[a,b]\cdot[c,c] = \begin{cases} [a\cdot c, b\cdot c] & \text{if } c = 2^{\omega} \;\wedge\; \#([a,b]\cdot[c,c]) \leq 2^{64} \\ [0, \texttt{MAX}] & \text{if } c = 2^{\omega} \;\wedge\; \#([a,b]\cdot[c,c]) > 2^{64} \\ [0,0] & \text{if } c = 0 \end{cases}$$

If none of the given cases occur, the engine cannot reason over satisfiability accurately anymore and throws an exception.

This kind of restriction seems to be heavy on first sight, however, most multiplication and division operations within Selfie are left and right shifts, respectively. Thus one of both factors is a concrete power of two.

An earlier approach was to execute multiplication with additions in an iterative way. The benefit is that the same cardinality check as for addition can be used and the power-of-two restriction may be abolished. Since most multiplications are left shifts, we noticed quite early that doing it in an iterative way is far too slow in terms of computation. For instance, shifting a value x ω bits to the left means $x \cdot 2^{\omega}$.

3.2.3.4 Division

For division, no cardinality check is necessary since dividing an interval indeed decreases the size of the resulting set. Meaning that no overshooting, thus no wrap around can be possible. However, a restriction for division exists as well due to *masking*. The divisor needs to be a concrete power of two if the dividend is a symbolic value. Overall, the purpose of masking is to remember which bits of a symbolic value already became concrete. That kind of information is essential when a symbolic buffer is used as a string, for instance. Masking is described further in [11].

Interval division is defined below.

$$[a, b] \,/\, [c, c] = \quad [a \,/\, c, b \,/\, c] \quad \text{if } c = 2^{\omega} \ \wedge \ c \neq 0$$

In order to be able to fully reconstruct the symbolic value when going backward, it is necessary to store the result, thus the quotient, as well as the remainder. Figure 12 gives an example of how trace generation for $[0, 255] \,/\, [8, 8]$ works.

pc	0x04	0x08	0x0C	0x0C
tc	-	-	-	12
lower	0	8	0	0
upper	255	8	7	31
state	sym	conc	sym	sym
tc	12	13	14	15

Figure 12: Trace generation of a division operation

Firstly, the remainder $[0, 7]$ is pushed on the trace, secondarily, the quotient $[0, 31]$, *i.e.*, the result. However, two entries are only pushed when the dividend is a symbolic value. Notice that those two entries have the same program counter entry, which helps later recognizing that an operation pushed two entries on the trace. Regenerating the symbolic value when going backward works simply by multiplying $[0, 31]$ to $[8, 8]$ and adding $[0, 7]$ finally.

3.2.3.5 Remainder

The remainder operator (%) is kind of cumbersome in interval arithmetic since it causes splitting an interval into two or more sub-intervals in some cases. As an example consider $[3, 6]\%[5, 5]$ which not only reveals $\{3, 1\}$ but rather yields $\{3, 4, 0, 1\}$ and therefore two intervals emerge: $[3, 4] \cup [0, 1]$. That is the case since the lower and the upper value, *i.e.*, 3 and 6, respectively, are not in the same residue class. To avoid such an issue, only specific cases for the modulo operation are allowed in our engine.

For the same reason as with division and due to the fact that for most remainder operations a concrete divisor is reasonable, only a constant for dividing an interval is permitted. However, all these restrictions are detectable at any time by the engine.

The legal cases in our symbolic execution engine are defined as:

$$[a, b] \,\%\, [c, c] = \begin{cases} [a \,\%\, c, b \,\%\, c] & \text{if } a \leq b \ \wedge \ a \,/\, c = b \,/\, c \\ [0, c - 1] & \text{if } a \leq b \ \wedge \ b - a \geq c \end{cases}$$

Wrapped-around intervals, which are described further in chapter 3.4, behave similarly complicated. Again sub-intervals emerge and therefore are wrapped intervals disallowed for the remainder operation by the engine.

There are two cases which last the interval entirely:

1. The lower and the upper value of the interval are in the same residue class. Then a usual modulo operation can be applied.

2. The interval already contains the whole residue class. Hence all possible residues are included in the resulting interval. This is called a *most ambiguous* interval [5], which is an interval that contains all possible remainders due to the operation. For instance $[3, 7] \,\%\, [5, 5]$, if one considers each value in the range solely and applies the modulo operation, the result would be $\{3, 4, 0, 1, 2\}$ which is equivalent to $[0, c - 1] = [0, 4]$.

In order to generate witnesses during backward constraining, the quotient is pushed before the ultimate result onto the trace as well. Similar to the division operation. Thus again two trace entries are needed.

3.3 Preparing Constraints

Symbolic values become primarily constrained when a relational operator such as <, <=, ==, !=, > or >= is involved. Meaning that only a subset of the interval remains after it has been constrained. Fortunately, it is only necessary to consider *less than* (<) since the compiler is able to convert all relational operators to a *less than* operation. Figure 13 denotes the conversion of the compiler, however, this is only true for unsigned integer arithmetic with wrap-around semantics.

>	`a > b`	⇔	`b < a`
<=	`a <= b`	⇔	`1 - (b < a)`
>=	`a >= b`	⇔	`1 - (a < b)`
==	`a == b`	⇔	`b - a < 1`
!=	`a != b`	⇔	`0 > b - a`

Figure 13: Conversion of relational operators by the compiler

Therefore, the instruction *Set on Less Than Unsigned* (`SLTU`) is sufficient. For reasons of simplification, we restricted the `SLTU` instruction in a way that a symbolic value can only get constrained by a concrete value for now. Hence, one operand has to be concrete if the other is symbolic. A relational operation on a symbolic value can lead to one of three different cases:

1. The symbolic value is strictly less or greater than the concrete value, hence only a `true` *or* a `false` branch needs to be considered further. For instance, $[0, 255] < [300, 300]$, the symbolic value is definitely less than the concrete value 300, thus the result is `true`. No constraining is necessary and no additional branch originates.

2. The concrete value is a *proper subset* of the symbolic value, hence a `true` *and* a `false` branch needs to be considered further, *e.g.*, $[0, 255] < [10, 10]$. The symbolic value can be constrained in two ways: Firstly, $[0, 9]$ and the whole expression becomes true, secondarily, $[10, 255]$ and the expression becomes false. Therefore, two branches have to be examined separately.

3. The concrete value is a proper subset of the symbolic value *and* the symbolic value is wrapped around. This holds in particular for == and != and is explained in chapter 3.4 in more detail. Three branches have to be considered, *i.e.*, either two `false` and one `true` branch or vice versa. As an example, $[0, 255] == [10, 10]$, the `true` constraint would be $[10, 10]$, the two `false` constraints $[0, 9]$ and $[11, 255]$, respectively.

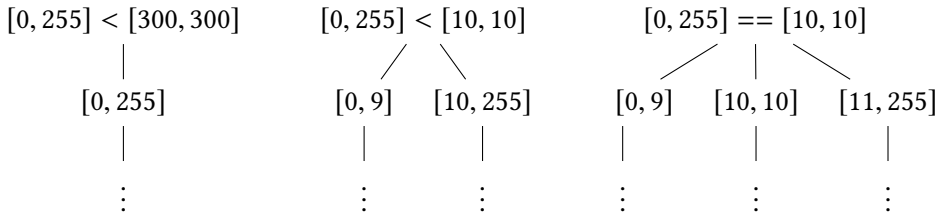

Figure 14: Possible branching cases triggered by relational operators

These constraints, however, are not applied during the forward pass but are prepared, *i.e.*, pre-computed and put on the trace as entries. The backward constraining pass then applies such a constraint in order to compute a satisfying assignment properly.

An SLTU instruction sets a trace counter pointing to $[1, 1]$ into the destination register `$rd` if the interval represented by source register one (`$rs1`) is less than the interval represented by source register two (`$rs2`), otherwise $[0, 0]$.

SLTU is in [13] defined as follows:

```
SLTU $rd $rs1 $rs2
```

Listing 6 concludes the first case: Source register one is strictly less or not less than source register two. This can be simply verified by checking (1) whether the upper interval bound of `$rs1` is less than the lower interval bound of `$rs2`, then `$rs1` is strictly less than `$rs2`, or (2) whether the lower interval bound of `$rs1` is greater or equal than the upper interval bound of register `$rs2`, which implicates the reverse case. This is sufficient since an interval is defined as monotonically increasing.

A new concrete trace entry, meaning that the lower and upper bound are equal, is generated by `setConcrete(x)`. The `saveState(tc)` call stores the passed trace counter, thus the `tc` that is carried by the destination register in this case, into the `tc` field and the program counter into the `pc` field of the latest trace entry, which was generated by the previous procedure call. Moreover, `updateRegState(reg, tc)` stores the passed `tc` into the passed register and increments the trace counter as well. The trace

entry is done, all references are set properly and everything is prepared to generate a new trace entry.

```
1  void symbolic_do_sltu() {
2    ...
3    // strictly less than
4    } else if (getUpperFromReg(rs1) < getLowerFromReg(rs2)) {
5      setConcrete(1);
6
7      saveState(*(registers + rd));
8      updateRegState(rd, tc);
9
10   // strictly not less than
11   } else if (getLowerFromReg(rs1) >= getUpperFromReg(rs2)) {
12     setConcrete(0);
13
14     saveState(*(registers + rd));
15     updateRegState(rd, tc);
16   }
17   ...
18 }
```

Listing 6: Verifies whether an interval is strictly less than another interval

The second case is more elaborated, constraints need to be computed and put on the trace additionally. References have to be set carefully in order to do not lose any information later when the opposite branch needs to be processed. Figure 15 shows how constraints are prepared by using the second case example of figure 14.

pc	0x04	0x08	0x08	0x08	0x08
tc	2	2	6	2	8
lower	0	10	0	0	1
upper	255	255	0	9	1
state	sym	cs	conc	cs	conc
tc	5	6	7	8	9

Figure 15: Trace generation of an SLTU instruction

The first trace entry shown in figure 15 is independent of SLTU, it might be from a *load* instruction beforehand. Assume that $rs1 contains the trace counter 5, thus represents the interval $[0, 255]$ and $rs2 represents $[10, 10]$. Further assume that the ultimate result should be stored in the same register as $rs1 is, *i.e.*, in this case is $rd = $rs1, which is quite common.

Firstly, the false constraint is computed, that is $[10, 255]$ in the example. The previous trace counter of that constraint is the same as from the unconstrained interval, therefore we simply adopt the tc field of the initial symbolic interval. The pc field receives the program counter of the current SLTU instruction and the state flag is set to constrained. Overall, this yields to the trace entry 6 in the example. Afterward, the

ultimate result, which indicates the `false` branch, is pushed on the trace. This is simply a concrete $[0, 0]$ interval and since we assumed that `$rd = $rs1`, $[0, 255]$ becomes overwritten by $[0, 0]$. More precisely, the trace counter in the destination register (`tc 5`) is replaced by the result. Therefore trace counter 7 is the new value carried by register `$rd`. To do not lose that kind of information, the trace counter of the constraint is stored in the `tc` field.

From that point on, a branch becomes independent of its initial symbolic value. As a sneak preview, imagine the algorithm wants to compute a satisfying assignment and goes backward. In this example, it follows the chain back from `tc` 7 to 6 and back to `tc` 2, but never again to `tc` 5. The point is that the algorithm omits the initial symbolic value $[0, 255]$ and never knows anything about it, which leads finally to two independent branches.

Secondarily, the `true` constraint is computed, pushed on the trace and the same procedure as before repeats. Ultimately, the destination register carries a trace counter which represents $[1, 1]$. That implicates that the first executed branch in the forward pass is always the `true` branch. Notice again the `tc` chain, which makes this branch independent of the initial symbolic value as well.

Listing 7 generalizes case two and shows how the true constraint is pushed on the trace.

```
1  void symbolic_do_sltu() {
2    ...
3    } else if (isConcrete(currentContext, rs2)) {
4      ...
5      // push true constraint: symbolic < concrete
6      setLower(getLower(tc_rs1), tc);
7      setUpper(getLower(tc_rs2) - 1, tc);
8
9      setStateFlag(CONSTRAINED, tc);
10
11     // point to previous tc of symbolic interval
12     saveState(*(tcs + tc_rs1));
13     updateRegState(rs1, tc);
14
15     // push true branch
16     setConcrete(1);
17
18     saveState(*(registers + rd));
19     updateRegState(rd, tc);
20   }
21   ...
22 }
```

Listing 7: Preparing the `true` constraint

Concluded, four entries are pushed on the trace rather than one. The `false` constraint, the `false` branch, which is the result of the `SLTU` instruction, followed by the `true` constraint and the `true` branch. The engine executes the `true` branch at first.

The third case involves a wrapped-around interval which is described in the following section. In principle, the interval becomes unwrapped, then the SLTU instruction falls back to case one or two.

3.4 Wrap-Arounds and Unwrapping

A wrapped-around interval is an interval where an over- or underflow occurred. For instance, add $[5, 5]$ to a full interval $[0, \texttt{MAX}]$, which yields $[5, 4]$. A wrapped-around interval can be easily recognized whether the lower interval bound is greater than the upper interval bound. It offends against the monotonically increasing property of our interval definition. Therefore the wrapped-around interval becomes *unwrapped*, which ultimately leads to two separate well-defined intervals again.

A wrapped-around interval is therefore defined as follows:

$$[a, b] = [0, b] \cup [a, \texttt{MAX}] \mid 2^{64} > a > b \geq 0$$

Unwrapping generates two valid intervals since $0 \leq b$ and $a \leq \texttt{MAX}$. However, two intervals rather than one need to be considered simultaneously, meaning that an additional branch originates. Figure 16 visualizes the definition of unwrapping.

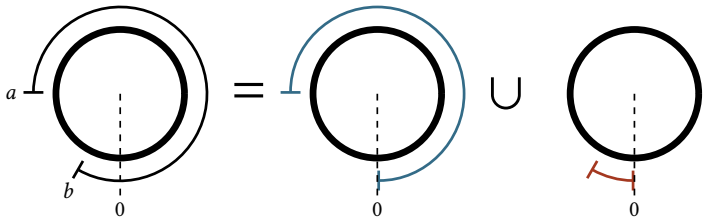

Figure 16: Unwrapping of a wrapped-around interval

Fortunately, it is sufficient to prove whether an interval is wrapped around at the same location where branches may originate for which the SLTU instruction is responsible. All other instructions, as well as arithmetic operations, can deal with wrapped-around intervals as they are. Except for REMU as described in chapter 3.2.3.5. Hence, a wrapped-around interval neither necessarily affects the forward nor the backward pass.

Listing 8 shows how a wrapped-around interval represented by source register `$rs1` becomes unwrapped. The procedure that implements the SLTU instruction is called recursively two times after unwrapping each part separately. Finally, listing 8 completes the third case outlined in chapter 3.3.

```
1  void symbolic_do_sltu() {
2    ...
3    if (getLowerFromReg(rs1) > getUpperFromReg(rs1)) {
4      // unwrap from lower to MAX
5      setLower(getLower(tc_rs1), tc);
6      setUpper(UINT64_MAX, tc);
7
8      // point to previous tc of symbolic interval
9      saveState(*(tcs + tc_rs1));
10     updateRegState(rs1, tc);
11
12     symbolic_do_sltu();
13
14     // unwrap from 0 to upper
15     setLower(0, tc);
16     setUpper(getUpper(tc_rs1), tc);
17
18     // point to previous tc of symbolic interval
19     saveState(*(tcs + tc_rs1));
20     updateRegState(rs1, tc);
21
22     symbolic_do_sltu();
23     ...
24     return;
25   }
26   ...
27 }
```

Listing 8: Significant part of unwrapping register `$rs1`

The references, *i.e.*, the respective `tc` fields of the two unwrapped trace entries, are set similarly as before with the constraints. Meaning that from that point on the two unwrapped, emerged intervals are completely independent of the previous wrapped-around interval.

A neat example for unwrapping a wrapped-around interval forms an expression including the == operator. Let us consider $[0, 255] == [10, 10]$ in greater detail. As mentioned earlier, the compiler converts an expression such as $a == b$ into $b - a < 1$, see figure 13, hence:

$$[0, 255] == [10, 10] \Leftrightarrow [10, 10] - [0, 255] < [1, 1] \Leftrightarrow$$

$$[10 - 255, 10 - 0] < [1, 1] \Leftrightarrow [-245, 10] < [1, 1]$$

Due to wrap-around semantics an underflow occurred and -245 actually is $2^{64} - 245$, or $10 - 255 \mod 2^{64}$. For simplicity, we keep the negative representation in the example, however, keep in mind that a negative value represents indeed a quite large positive value. This is where unwrapping comes in since the lower interval bound (-245) is greater than the upper interval bound (10).

Two independent branches need to be considered further:

1. $[0, 10] < [1, 1]$
2. $[-245, \texttt{MAX}] < [1, 1]$

In the first case, the concrete value is a proper subset of the symbolic value and hence two constraints have to be computed. In the second case, the symbolic value is strictly not less than the concrete value, which yields a `false` path and no constraint is necessary. Summarized, we obtain three different branches: (1) a `true` branch with symbolic value $[0, 0]$, (2) a `false` branch with symbolic value $[1, 10]$ and (3) another `false` branch with symbolic value $[-245, \texttt{MAX}]$, whereas `MAX` is equal to $2^{64} - 1$. In order to stay in the same notation, we substitute -1 for `MAX`.

The backward pass is able to compute a satisfying assignment by inverting all instructions backwardly. In the example, the subtraction that the compiler created needs to be inverted. The subtraction instruction computed during the forward pass:

$$[10, 10] - [a, b] = [10 - b, 10 - a]$$

In order to obtain the inverted value, it is sufficient to plug in a and b into the right-hand side [11]. Therefore, our three satisfying assignments are:

1. $[10 - 0, 10 - 0] = [10, 10]$
2. $[10 - 10, 10 - 1] = [0, 9]$
3. $[10 - (-1), 10 - (-245)] = [11, 255]$

As expected, the satisfying assignments match with them in figure 14. Figure 17 below denotes the corresponding trace entries.

pc	0x04	0x08	0x08	0x08	0x08	0x08	0x08	0x08
tc	2	2	6	2	2	9	2	11
lower	-245	-245	0	0	1	0	0	1
upper	10	MAX	0	10	10	0	0	1
state	sym	sym	conc	sym	cs	conc	cs	conc
tc	5	6	7	8	9	10	11	12

Figure 17: Trace generation of an SLTU instruction including unwrapping

Trace entries 6 and 8 has the subroutine, which is responsible for unwrapping, created, see listing 8. The ultimate outcomes of the SLTU instruction are provided by trace entries 7, 10 and 12, respectively.

4 Conclusion

This work mainly presented the trace data structure, interval arithmetic, the importance of constraints and issues concerning wrapped-around intervals. The design of the trace allows backtracking of code execution, which is essential for finding satisfying assignments as well as unsatisfiable paths. The data structure in combination with interval arithmetic leads to higher performance over using an SMT solver but also restricts the set of programs that can be executed by the engine. A trade-off between performance and completeness has arisen.

Problems that symbolic execution engines are trying to solve are challenging. Maintaining correctness and soundness is particularly hard. Our engine is far away from being complete, which might be improved incrementally in future work, for instance, by tackling some of the restrictions. In particular, the limitation that for almost every arithmetic operation at most one operand is allowed to be symbolic. A more dynamical approach may be interesting, especially for addition and subtraction.

Further problems are `while` loops, however, most symbolic execution engines are struggling with the path explosion problem. Sufficiently strong invariants, as well as strict assumptions, may at least partially solve such problems in our engine.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *CoRR*, abs/1610.00502, 2016.

[2] Simon Bauer. Symbolic Execution with Selfie: Systems. 2018.

[3] Nikolaj Bjørner. SMT Solvers: Foundations and Applications. In *Dependable Software Systems Engineering*, pages 24–32. 2016.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[5] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance is Bliss. *ACM Trans. Program. Lang. Syst.*, 37(1):1:1–1:35, 2014.

[6] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval Arithmetic: From Principles to Implementation. *J. ACM*, 48(5):1038–1068, 2001.

[7] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[8] Christoph Kirsch. *Selfie: Computer Science for Everyone.* Leanpub, 2017.

[9] 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, October 23 - 27, 2017*, pages 198–213, 2017.

[10] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[11] Sara Seidl. Symbolic Execution with Selfie: Logics. 2018.

[12] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. v1.2.

[13] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

List of Figures

Listings