

Bachelor Thesis

Symbolic Execution with SELFIE: Systems

Simon Bauer
simon.bauer@cs.uni-salzburg.at



Department of Computer Sciences
University of Salzburg, Austria

18. August 2018

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

Abstract

This thesis presents a simple symbolic execution engine, which was integrated into the **SELFIE** project, an educational platform already containing a self-referential compiler, emulator and hypervisor. The idea behind symbolic execution is to provide a program with arbitrary symbolic - rather than concrete - input, which allows the symbolic engine to efficiently explore all control flow paths of the program. Our engine differs from state-of-the-art solutions as it does not utilise a constraint solver and is therefore faster than other tools. As a consequence, the engine is only able to handle a subset of all possible programs though. The thesis gives an overview over the engines building parts and their differences as well as similarities compared to other symbolic tools.

Contents

1	Introduction	3
1.1	SELFIE	3
1.2	Symbolic Execution	3
1.3	Problem Definition	4
1.4	Capabilities	4
1.5	Contribution	4
2	Building Blocks of the VIPSTER Engine	6
2.1	Symbolic Values	6
2.2	Exploring Control Flow	10
2.3	Representing Execution State	12
2.4	Deciding Satisfiability	14
2.5	Symbolic Strings	17
2.6	Environment Access	19
2.7	Operation Breakdown	20
2.8	Runtime Improvement	23
3	VIPSTER in Action	25
3.1	Paths	25
3.2	Equality	26
3.3	Loops	28
3.4	Working with Strings	29
4	Conclusion and Related Work	33
5	Appendix	34

1 Introduction

1.1 SELFIE

The SELFIE Project features a compiler (`starc`), emulator (`mipster`), hypervisor (`hypster`) and a tiny library (`libstar`). The compiler accepts `C*`, a small but powerful subset of `C` and translates it to `RISC-U` machine code, which is a subset of the `RISC-V` instruction set. Selfie is implemented in a single 10.000-line source file of `C*`-code and is completely self-referential. This means that the compiler can translate all of selfie into a `RISC-U` binary which can then be executed by a `mipster` instance or a virtual machine provided by `hypster`. Everything within selfie is kept simplistic because it is not only used for research but also for teaching with focus on compiler construction, operating systems and virtual machine monitors [7, p.1]. Selfie was built and is maintained by the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg.

1.2 Symbolic Execution

Every programming language provides manipulable data containers. In `C*`, one can declare variables and use them to store data. During normal execution, such variables contain concrete values that influence the program's control flow. Testing code with random or even specific concrete input always comes with the risk of missing some possible control flow paths and therefore also the errors that may hide there. When symbolically executing programs, the data containers are supplied with symbolic values which represent arbitrary data. The various control flow paths of a program are then explored for whole classes of input. Hence a symbolic engine considers all possible input values and surely evaluates all paths. The engine also records how the values are modified and under which assumptions certain paths were taken. With this information, it is possible to reason about whether or not some code is actually reachable in practice and if certain properties about the input hold. Detectable errors include buffer-/numerical overflows, divisions by zero and more. In case some problem is found, a concrete witness can be generated that will lead to the flaw during concrete execution[6, p.1,2].

Symbolic Execution can be classified as a compromise between concrete, data-driven testing and program proving. Modern engines can achieve high code coverage and are already heavily utilized in practice to test software [4, p.1].

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

The engine takes **RISC-U** machine code as input and decides if parts of this code may actually ever be executed. In particular, it goes through all runs of the program and for each one, decides if there exists an input that results in this very run. If such an input exists, its actual value is reported, which is going to lead the program through this run during normal execution.

Since **VIPSTER** does not use a constraint solver, it can execute only a subset of all possible **RISC-U** programs. In particular all binary arithmetic operations are restricted to have at most one symbolic operand. Furthermore, multiplications are only allowed by powers of two in order to properly detect overflows. Character and string support is very rudimental due to **VIPSTERs** 64 bit interval design. Lastly, the memory cannot be accessed via symbolic values as pointers.

1.5 Contribution

VIPSTER, the symbolic execution engine within **selfie**, was implemented by Sara Seidl, Manuel Widmoser and Simon Bauer. Sara coded our core data structure as well as advanced memory techniques such as memory synching and masking of values. She also implemented most of the backwards reasoning process. Manuel took care of the interval arithmetics of the engine, including also detecting and handling overflows. The mechanic that handles and prepares constraints on the trace was also mostly done by him. I was responsible for the testing, memory bootstrapping and the environment handling as well as some experimental prototyping. All features were planned and discussed as a group in advance.

Every one of us wrote a thesis, describing a specific part of the engine. Sara focuses on the logics and satisfiability capabilities [8] while Manuel concentrates on the arithmetics and data structures [9]. This piece is aimed

at giving a more general view over the parts of the engine, the way they work together and how our approach differs from other state-of-the-art symbolic execution engines.

Lastly, our advisor Christoph Kirsch and also the other members of the Computational Systems Group provided us with valuable input throughout the development of VIPSTER.

2 Building Blocks of the VIPSTER Engine

VIPSTER differs from traditional designs since we wanted to keep it lightweight like the rest of SELFIE. At its core, the engine interprets the machine code just like the `mipster` emulator, but there are of course more components built around that. This section presents the building blocks of VIPSTER and compares them to other popular design choices.

One famous open source engine is KLEE, which was published in 2008 [5]. It is used for comparisons throughout this section.

2.1 Symbolic Values

Symbolic values are placeholders for whole classes of concrete values. Such a class contains the values that may occur in place of the symbolic value during normal execution. Most engines encode these classes indirectly as logical formulas [2, p.24]. But for simplicity reasons, it is assumed that such a class is just a set that contains the concrete values. Since VIPSTER performs symbolic execution on machine level, each register or memory possibly contains a symbolic value. Hence each register or memory location logically holds such a set of concrete values. This information needs to be handled somehow inside the engine. For a symbolic value x let $S_x := \{a \in D \mid a \text{ can occur instead of } x\}$ denote this set. The Domain D is the set of available values and is restricted by the machine. Since SELFIE supports only unsigned 64-bit integers [1], $D := \{a \in \mathbb{N} \mid 0 \leq a \leq 2^{64} - 1\}$. There are two types of operations commonly performed on symbolic values:

Constraining Symbolic Values

A symbolically executed program may contain conditional statements involving symbolic values. When a path of the program is executed, symbolic values have to comply to these conditions [6, p.385]. So these statements constrain the set of concrete values that a symbolic value represents. Moreover, only the elements for which the constraint holds remain in the set. Let x be some symbolic value in the program representing the set S_x . If a part of the program is executed under the assumption $\mathcal{A}(x)$, then the constrained symbolic x^c represents the set $S_{x^c} = \{a \in S_x \mid \mathcal{A}(x) \text{ holds}\}$. Those constraints are called the “path condition” - the assumptions under which the current part of the program is executed. This condition is initially empty and gets extended as execution carries on [6, p.386].

Arithmetic Operations With Symbolic Values

SELFIE supports several basic unary and binary arithmetic operations. For binary operations with two symbolic operands, both operands logically represent a set of concrete values. These two sets need to be merged properly, which can be quite challenging. In some cases, our set representation isn't even sufficient for the result of such operations. Hence for simplicity reasons, VIPSTER only supports operations where only one operand is symbolic. When such an arithmetic operation is performed on a symbolic value, the operation is also performed on each element of the concrete set as well. Let again x be a symbolic value with the respective set S_x . The new symbolic value $x' = f(x)$ for some operation f represents the set $S_{x'} = \{f(a) | a \in S_x\}$. Arithmetic operations are also considered part of the "path condition". They are added to the condition in the form of "symbolic formulas" - arithmetic expressions over symbolic values [6, p.386].

So both constraining and operating on symbolic values alters their respective set of concrete values. This information has to be represented inside of the engine.

Path Condition and Symbolic Trees

KLEE stores all constraints and arithmetic operations as one path condition. This formula can then be checked by a constraint solver - which is discussed in a later chapter. Additionally, for each symbolic value exists a binary tree. The tree records the arithmetic operations performed on the symbolic value. Such a structure is initialized as a single concrete node or symbolic marker and grows during execution. [5, p.4]

Assume an arithmetic operation with two operands, each represented by a single node or tree. The result of this operation is a tree as well. Its root is a node with detailed information about the performed operation. The left and right children of the root node are the original trees of the operands. Figure 1 shows such a symbolic tree for a statement $2 * (7 + s)$, where variable s is considered to be symbolic. The addition of 7 and s forms the subtree with the root ADD. Since the multiplication is evaluated afterwards, it takes the additions result as the second operand. This result depends on s though and is not known yet, hence the whole addition-subtree serves as operand for the multiplication. Every memory location of a program and each register points to a tree structure [5, p.4]. Hence for each stored value, it is remembered how it was constructed by the program. Our first prototype also used this design to represent symbolic values.

Listing 1 shows how the implementation of unsigned addition handles trees. If at least one operand is symbolic, a new root node is created (9). This

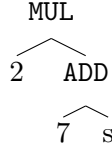


Figure 1: Symbolic Expression Tree

node is tagged with the **ADD** operation and references the operand trees as its children. In case both operands are concrete, a new concrete node with the result is created right away (6,7,8). This is also done in KLEE and ensures that registers such as the **frame pointer** or **stack pointer** always point to a single concrete node [5, p.4].

```

1 uint64_t con; uint64_t n; uint64_t * symNode; con = 0;
2     if (SYMBOLIC.CON == getSymNodeType(*(registers+rs)))
3         if (SYMBOLIC.CON == getSymNodeType(*(registers+rt)))
4             con = 1;
5 if (con) {
6     n = getSymNodeValue(*(registers+rs)) + getSymNodeValue(*(
7         registers+rt));
8     symNode = createSymbolicNode(SYMBOLIC.CON, n, (int*) 0, (int*)
9         0);
10 } else
11     symNode = createSymbolicNode(SYMBOLIC.OP, FCT_ADD, *(
12         registers+rs), *(registers+rt));
13 *(registers+rd) = symNode;
14 pc = pc + INSTRUCTIONSIZ;

```

Listing 1: ADD with symbolic trees

So KLEE stores both the constraints and arithmetic operations as a path formula. The concrete values are indirectly contained within the path condition here. Moreover, there might be dependencies between variables of the formula. Hence - as already stated - the set representation isn't even sufficient in this case.

This makes it seem that the symbolic trees are not actually useful, but imagine some variable **x** in a method that is dependant on symbolic input. The input is probably modified until it reaches the method. The symbolic tree of variable **x** contains all those modifications. So once a possible concrete input is found, the tree allows for constructing the value of **x** without executing the whole program again. However, both the trees and the path condition grow as the execution continues, requiring more memory over time. Although this is possible to handle, we would favour a solution that does not require dynamic memory allocation at all.

Intervals

KLEE maintains the concrete values indirectly via the path condition. However, extracting the values from there is done by a solver and quite costly [5, p.5][4, p.88]. Therefore we tried to find some other compact way of representing the set of concrete values. More important, the constraints on the set and the arithmetic operations should be applicable on the fly. Hence the final VIPSTER engine represents symbolic values with intervals. An interval is a tuple $[l, u]$ for $l, u \in [0, 2^{64} - 1]$ where l denotes a lower and u an upper bound. These serve as bounds for concrete values that may be used instead of the symbolic placeholder. As with trees, each memory location still points to one interval. Concrete values can be represented by an interval with two equal bounds. Usually the lower bound of an interval is always smaller or equal to the upper bound. In some cases, however, VIPSTER generates intervals such that the lower bound is actually greater than the upper one. This can happen because of arithmetic overflows or as part of the reasoning process. The interval design is of course quite restricting since it is assumed that a set of concrete values contains contiguous elements [9][8]. As later examples will show, it is still sufficient though for symbolically executing non-trivial code.

The SELFIE compiler targets the RISC-U instruction set [1]. The only comparison instruction included in RISC-U is the SLTU (*set less than unsigned*) instruction. Hence the compiler converts all common comparisons in the C* source code into expressions containing “less than”. Therefore VIPSTER only has to support the SLTU instruction, although with the restriction that only one operand can be symbolic [9]. In this case, one interval bound can be adjusted to satisfy the constant constraint. The actual implementation is still quite tricky as the engine has to handle arithmetic overflows here [9].

Any arithmetic operation combines the intervals of its operands to new intervals. Listing 2 shows the respective implementation of unsigned addition. In essence, both lower and upper bounds are added each $(4, 5)$. Assume a number from within $[l_1, u_1]$ is added to another one bounded by $[l_2, u_2]$. The result is at least $l_1 + l_2$ and at most $l_2 + u_2$. The actual arithmetics are more complex though, to properly reflect the 64-Bit wrap-around semantics of a RISC-machine [9].

```

1 if (rd != REG_ZR) {
2     saveState(*(registers + rd));
3     if (cardinalityCheck(*(registers + rs1), *(registers + rs2))
4         ) {
5         setLower(getLowerFromReg(rs1) + getLowerFromReg(rs2), tc
6             );
7         setUpper(getUpperFromReg(rs1) + getUpperFromReg(rs2), tc
8             );
9     } else
10        setMaximum();
11    setStateFromReg(rs1, rs2, tc);
12 } else
13    clearTrace();
14 pc = pc + INSTRUCTIONSIZE;
15 updateRegState(rd, tc);

```

Listing 2: ADD with intervals

This design does not require any dynamic memory allocation, still each instruction has to be adjusted. Intervals represent the set of possible concrete values in a very compact way, requiring only two numbers per symbolic value. Since every virtual memory address and each register points to exactly one interval, the required memory can be allocated one-time when the engine is initialized. Intervals also allow for simpler reasoning without a constraint solver, since the set of concrete values is always known [8]. The simplicity of intervals comes with restrictions though - some are discussed in later chapters. As already stated, only one operand of an arithmetic operation can be symbolic. Additionally, the engine only supports a subset of all the arithmetic operations possible in C* [9].

2.2 Exploring Control Flow

All the different possible control flow paths of a program form the so called execution tree [6, p.387]. For a simple Code-Snippet 9, one can construct the tree displayed in Figure 2.

```

1 uint64_t s; // symbolic value
2 if (s < 11)
3     if (s > 4)
4         return 0;
5 else
6     return 1;
7 return 2;

```

Listing 3: Simple Path Tree (Code)

The root of such a tree represents the entry point of execution. Each leaf is an event where execution stops, which can result from a normal program

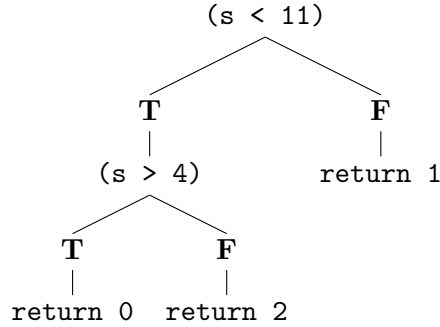


Figure 2: Simple Path Tree (Visualization)

exit or a crash. As shown later, **VIPSTER** records all the changes made to a programs state. Hence it can also happen that the engine itself runs out of memory and can't execute the program any further. A branching in that tree represents **IF** or **WHILE** constructs, which direct the control flow. One path from the root to a leaf represents one possible run of the program. The symbolic engine should explore all paths contained in the tree. Because of the tree structure, graph algorithms are commonly used for this.

Breadth-First-Search

Professional tools often use a breadth-first-search combined with certain heuristics [2, p.8,9]. The engine maintains a set of program states and initially, this set only contains a single state - the entry point of the program. Each state is executed for a few instructions and then exchanged with another one. When a branching is reached, the current program state gets copied once for each possible control flow choice and added to the set. **KLEE** favours paths that have not been executed recently and include previously unexecuted code [5, p.6,7], while other tools focus on paths that previously uncovered errors [2, p.9]. With this design an engine needs to store and manage several program states though, making it not suitable for our simplicity approach.

Depth-First-Search

VIPSTER implements a depth-first-approach. The program is executed from the entry point. At each branching, one control flow path is taken arbitrarily and the engine remembers the other possible choices. Upon reaching an exit point of execution, the engine goes back to the most recent branching and picks a different path. This requires a data structure that allows undoing changes made to the state of the program - it is presented in the next section.

VIPSTER needs to store only one program run at a time with this design. Exploring loops or recursions may, however, lead to an infinite execution of the given path in case the condition includes a symbolic value [2, p.16][4, p.85]. For each iteration the engine would also consider the case that the condition yields true, exploring yet another iteration. It should be noted, that not every loop or recursion is guaranteed to result in an infinite path. Different loop scenarios are discussed in the examples-section.

Path Explosion

VIPSTER and all other modern symbolic execution engines face the problem of path explosion. More complicated programs tend to have huge numbers of paths to explore. Scenarios like loops or symbolic memory accesses are particularly prone to generate many new paths. Most tools rely on invariants and smart symbolic memory modelling to stem this problem. VIPSTER in particular does not allow memory access with symbolic pointers. [4, p.87][2, p.16,17]

2.3 Representing Execution State

The engine needs the capability of remembering the state of a symbolically executed program. Both execution strategies - BFS and DFS - require this. The state of a program is defined by its virtual memory and the set of registers. In the symbolic context this also includes the symbolic values (trees or intervals) that are contained in memory locations and registers plus the path condition. Hence, in KLEE a state holds the constructed path condition as well as the symbolic trees [5, p.5]. VIPSTER implicitly remembers the path conditions within the intervals.

The predecessor of KLEE used native OS **forking** to handle multiple states [3, p.3]. Though to improve performance, KLEE implements its own context structure with several optimizations. For example, memory objects are tracked, enabling copy-on-write semantics at object-level. This greatly decreases the memory required per state. [5, p.5]

Since VIPSTER explores the execution tree with depth-first search, there is no need to store multiple states simultaneously. Only one single program run is constructed at a time - from the root of the execution tree to an endpoint of execution. To obtain another run, VIPSTER backtracks to a previous branching and executes another path from there. Hence a program run has to be recorded in a way that enables reverting to a previous state of that run. The solution to this is a trace data structure that records the effect of each executed instruction. Assuming that for an executed instruction, its effect on the state is known. Then this effect can be reverted to obtain

the state of the program right before the instruction was executed. This benefits from the fact, that each RISC-U instruction modifies at most one arbitrary register or memory location and the **program counter**. So when an instruction changes the state of the program, it may override at most one value inside a register or memory with a new value and update the **program counter**. It is therefore sufficient to remember for each executed instruction the respective program counter, the value that was overwritten and the new value. With this information, VIPSTER is able to essentially construct a versioning of all states that were present during a single execution run.

The trace is implemented using several arrays, one for each needed information. VIPSTER maintains a bump pointer, the so-called **trace counter** (*tc*), to remember where to insert new entries. In detail, one entry contains the following information:

- **program counter** of the executed instruction
- **trace pointer** of the overwritten value
- **lowerBound** of new interval
- **upperBound** of new interval
- **state** (concrete/symbolic/unsatisfiable)

The overwritten value is referenced by a trace index, which points to the trace entry where the old value was generated. There is no need of remembering the old **program counter** since the previous entry implicitly contains it. Furthermore the **program counter** links to the executed instruction itself in the binary. From there the manipulated register or memory location can be extracted, hence it is not explicitly referenced within the entry. A modified register or memory itself only contains a trace index pointing to the respective entry, that represents the modification.

The trace contains all the intervals ever generated during one program run and therefore enables VIPSTER to revert to a previous program state. In order to do so, the trace is “executed” backwards and for each instruction, the overwritten value from the **trace pointer** is restored. In general, the trace also contains additional entries with information about which paths were already taken [9] and also details for the reasoning process [8].

Symbolically executed programs often contain data and strings at the data segment as well as arguments that are placed on the stack by the operating system. This data has to be inserted into the trace before the execution starts so that the program can properly interact with it.

2.4 Deciding Satisfiability

Most symbolic execution engines collect a path condition along the executed program run. This condition consists of all assumptions, under which the current path was taken. At some point a constraint solver is consulted and decides whether this formula is actually satisfiable or not. [4, p.84] Moreover, in case it is satisfiable, the solver provides a concrete witness - a satisfying assignment to the formula. This directly translates to the reachability of the executed path. When the solver decides the path condition is not satisfiable, then there is no possible input that could ever lead to that execution path. If on the other hand, the solver yields that the condition is satisfiable, then there exists some input that can lead there - moreover the solver even provides such an input. The problem of reachability of code is thereby reduced to deciding the satisfiability of a logical formula. So the set of formulas that are decidable by the solver restricts the class of programs that are executable by the engine.

State-of-the-art engines follow some heuristics on when in particular to ask the solver to check the path condition [2, p.26]:

Eager Evaluation

The path condition is checked at each branching of the execution tree. Some paths may turn out to be unsatisfiable so they don't have to be explored unnecessarily. Asking the solver at every branching is quite costly as the decision process can take a long time.

Lazy Evaluation

The solver is queried as little as possible. Hence some paths may be explored despite they are not reachable. However, less time is spent checking the path condition over and over again.

Even today, the solver is the performance bottleneck of any modern symbolic engine [4, p.88]. The core idea behind **VIPSTER** is to utilise some simpler reasoning mechanic. As stated before, intervals are used to represent sets of concrete values. The representation is already restricted since it is assumed that such sets can only contain contiguous elements. Therefore **VIPSTER** logically can only decide a subset of the formulas, usually decidable by modern solvers. Hence also the class of executable programs is restricted in comparison to modern engines. Though in return the reasoning process is simpler and faster than solving constraint formulas.

Assume a program is executed which is not in the class of supported programs. Let p_1, p_2, \dots, p_k be all k possible execution paths of that program.

Then $r_1, r_2, \dots, r_k \in \{R, U\}$ denote for each path p_i ($i \in [0, k]$) if it is actually reachable ($r_i \equiv R$) or unreachable ($r_i \equiv U$). Hence the symbolic engine provides some r'_1, r'_2, \dots, r'_k as a result. For each r'_i ($i \in [0, k]$) one of the following cases may hold:

- Correct result: If $r'_i \equiv r_i$ holds.
- Incorrect result:
 - Incompleteness:
The engine concluded that the path p_i is reachable $r'_i \equiv R$ but it actually is unreachable $r_i \equiv U$. This is a false positive error because a potential problem is discovered where there actually is none. Incompleteness is to some extent acceptable if the occurrence rate is not too high.
 - Unsoundness:
The engine concluded that the path p_i is unreachable $r'_i \equiv U$ but it actually is reachable $r_i \equiv R$. This is a false negative error because a potential problem was not uncovered. Unsoundness is not acceptable and should therefore be detected by the engine. VIPSTER can observe when execution may become unsound and will then abort executing the program.

VIPSTER represents symbolic values through intervals. As already discussed, constraints and operations directly update these intervals. Hence the set of concrete values is always known and there is no need for a solver. Still, the interval handling introduces its own problems:

Aliasing

In the symbolic context, aliasing means that several variables contain the same symbolic value. Hence those variables are just aliases for the one value. Listing 4 shows an example where this problem occurs. Assume variable **s** is initialized with some symbolic value. The statement in line 3 assigns this symbolic value to the variable **a**. Both variables then are constrained in lines 5, 6, **s** to be less than 10 and **a** to be greater than 10. They both refer to the same symbolic value though, hence there is no value that is both smaller and greater than 10. The assignment in line 3 just copied the interval from **s** over to **a**. Therefore, later on, VIPSTER does not know that these intervals are related. Hence the path, returning 0, would be reachable from VIPSTER's perspective.


```

1 uint64_t s;
2 uint64_t a;
3 a = s;
4
5 if (s < 10)
6     if (a > 10)
7         return 1;
8 return 0;

```

Listing 4: Simple Aliasing Example

The core problem lies in the fact, that the data flow of a symbolic value can divide. It must be remembered that such diverging data flows are still linked to each other.

Witness Generation

Assume **VIPSTER** executes a reachable path and properly updates intervals along the way. In the end, the intervals represent the state of the symbolic values at the endpoint of execution. The engine should then output a concrete input, which will lead to the execution of the path. This would require knowing the states of the symbolic values at the start of the execution though. Since the symbolic values are most likely modified during execution, this initial state cannot easily be obtained. Thus **VIPSTER** can only tell with which concrete values the execution of a path ends, but not with which values it has to start to reach the current path.

To solve the **Witness Generation** problem **VIPSTER** goes through the trace backwards and propagates the applied constraints to the beginning to obtain witnesses. So for each **RISC-U** instruction, there exists a *backwards*-counterpart which takes the constrained solution and modifies the operands to meet the constraints. These instructions tend to be quite complicated in general [8]. Assume a variable is initialized with the symbolic value $[0, 10]$, incremented and then constrained by < 10 , just like in Listing 5. The increment results in the interval $[1, 11]$, which is then constrained to $[1, 9]$. After reaching the endpoint, **VIPSTER** starts to propagate the constraints backwards. In particular, the **ADUU** instruction for the increment in line 2 now performs a decrement, giving the interval $[0, 8]$ which is also the correct witness. It should be noted, that the backwards propagation also changes the state of the program. These changes need to be reversible as well. Therefore the trace is both executed backwards and extended forwards - which is discussed in more detail later on.

```

1 uint64_t s;
2 s = s + 1;
3
4 if (s < 10)
5     return 1;
6 return 0;

```

Listing 5: Simple Backwards Example

Once a path is recorded onto the trace, **VIPSTER** propagates the constraints backwards to generate witnesses. This actually also fixes the **aliasing** problem! The previously diverging data flows now nicely meet up, as the program is executed backwards. This requires some attention when backwards-executing LD load instructions. In essence, a backwards-load then stores a value from a register into memory. Both the register and the memory location may contain an already constrained symbolic value. These two intervals must be handled properly, which is called memory synching [8]. The value already in memory was constrained and stored there before. The value in the register was also constrained and should be stored into memory. Hence the resulting symbolic value should satisfy both constraints, meaning the intersection of those intervals is stored into memory. When this intersection is empty, the symbolic value is unsatisfiable and the current path unreachable.

So **VIPSTER** is able to generate concrete witnesses even without a solver. The satisfiability checking process only requires the engine to go through a program run a second time [8]. The propagation process is carried out until for each symbolic values either a witness is found or it is proven unsatisfiable.

2.5 Symbolic Strings

Memory in **SELFIE** is byte-addressed and word-aligned, where each word is 64 bits [1]. Strings are just 8-bit characters stored contiguously in memory, so each memory word contains 8 characters. Usually, the actual strings are stored in memory and pointers are used to access them. But with intervals and the trace, there is some more indirection involved.

Listing 6 shows how a string literal is assigned to some pointer, Listing 7 shows the respective minimalist assembly. The **SELFIE** compiler puts the actual characters of “Hello World!” at the data segment of the binary [1]. Line 1,2 of the assembly shows how the pointer to this memory is calculated - relative to the **global pointer**. Assuming that **str** is the only local variable, it is located on the stack at address **\$fp-8**. In line 3 of the assembly, the pointer is stored at that local variable.

tc	1	2	3	4	5	6	7
\$pc	0x0	0x0	0x0	0x0	0x0	0x4	0x8
prev tc	0	0	0	0	0	5	0
lower	1819043144	560229490	0xFFC	0x024	-24	0x00C	0x00C
upper	1819043144	560229490	0xFFC	0x024	-24	0x00C	0x00C
state	con	con	con	con	con	con	con

Figure 3: Strings in VIPSTER: Trace

```

1 uint64_t * str;
2 str = (uint64_t *) "Hello_World!";

```

Listing 6: Strings in VIPSTER: Code

```

1 addi $t0,$zero,-24
2 add $t0,$gp,$t0
3 sd $t0,-8($fp)

```

Listing 7: Strings in VIPSTER: Assembly

Figure 3 displays the generated trace when symbolically executing the above instructions. For this example, the values of the global and frame pointer are just placed inside the trace at entry 3 and 4. Entries 1 and 2 show the actual string “Hello World!” in numerical representation, stored in two separate machine-words. VIPSTER initially copied the string there from the data segment to make it trace-compliant. The entries 5, 6 and 7 recorded how the pointer to the string was calculated and stored at the local variable. To obtain the string, several references need to be resolved:

1. The variable `str` at `$fp-8` contains the trace index 7.
2. Entry 7 yields that the pointer actually points to memory address `0x00C ($gp -24)`.
3. And memory address `0x00C` contains the trace index 1.
4. Entry 1 finally contains the first 8 characters of the string as a concrete interval.

This indirections need to be resolved for example when VIPSTER itself has to access the string for handling system calls.

2.6 Environment Access

Symbolically executed programs interact with the environment through system calls [2, p.14,15,16]. There arise several challenges with properly handling these calls during symbolic execution. System calls do not only change the state of the program but some also change the state of the environment. So for all executed paths, the environment states have to be separated. This could be done by either managing several copies of the environment (for a BFS path exploring approach) or recording and reverting the environment state (for a DFS strategy). For simplicity though, VIPSTER does not monitor the environment in any way.

SELFIE features five system calls: `open`, `read`, `write`, `malloc` and `exit`. Both `exit` and `malloc` are fairly simple to handle. An `exit` call just initiates the backtracking process, independent from whether the exit code is symbolic or concrete. For a `malloc`, it is verified that the requested amount of memory is a concrete value, the memory is then allocated and a pointer gets returned. Allowing for symbolic amounts would increase the complexity since each possible concrete request needs to be executed as a separate path [2, p.10]. In SELFIE, memory allocation is handled with a stack allocator and a bump pointer [1]. Allocated memory is actually never freed even when the programs state is reverted on the trace. Hence when some memory is requested during the execution of some path, that memory stays allocated even when the path isn't stored on the trace anymore. This is very inefficient but is done so for simplicity at the moment.

Open

```
uint64_t open(uint64_t filename, uint64_t flags, uint64_t mode);
```

The `open` syscall provides access to a file in the file system via a file descriptor. A valid path to a file must be provided as an argument to the call and the program can then use the returned descriptor to read or write data. The path string may be concrete or symbolic. VIPSTER just processes the lower bounds of the characters intervals as the path. Hence the string is made concrete, and then validated. If the converted string matches a file, a proper descriptor is returned. KLEE can even handle fully symbolic paths by then providing symbolic files and buffers. Furthermore, the handling of other system calls can be customized and adapted to specific needs. This is done with models that are written in normal C code [5, p.7].

Read

```
uint64_t read(uint64_t fd, uint64_t buffer, uint64_t bytesToRead);
```

`read` calls actually don't change the environments state since they only request information. Nonetheless, they do depend on the state of the environment at the exact time the call is performed. The goal of symbolic execution is to execute all execution paths of a given program. Hence when a program reads a single character from some file descriptor then the result should be a symbolic value representing all validly obtainable characters. This is done even if the file descriptor refers to a real file. Hence, **VIPSTER** handles `read` calls completely detached from the environment.

The actual only way of generating symbolic values is through `read` calls. In Listing 8 a pointer `s` is declared and assigned with 1 byte of heap memory. Then one character is read from the console to this buffer. **VIPSTER** will put the interval `[0,255]` at the virtual address. This is because in **SELFIE** each character is 8 bits long, allowing for representing each integer between 0 and 255. There is no interaction with any file so the file descriptor 0 is passed to the `read` call. Actually any value can be used since the file descriptor is never validated.

```

1 uint64_t main(uint64_t argc, uint64_t*argv)
2 {
3     uint64_t * s;
4     s = malloc(1);
5     read(0, s, 1);
6 }

```

Listing 8: Generating Symbolic Values

Write

```
uint64_t write(uint64_t fd, uint64_t buffer, uint64_t bytesToWrite);
```

`write` calls change the environments state, but these changes are not tracked by **VIPSTER**. Since the engine only allows for opening concrete files, it makes sense to let a `write` call operate on the real file descriptor. As with file paths, the actual characters are then converted to concrete values by simply taking only the lower interval bounds and writing them to the file.

There are more complicated scenarios, for example symbolically executed programs may use files to store data that they access later again. Professional tools of course offer solutions for such cases, too but **VIPSTER** only handles the most basic system calls. [2, p.14,15,16]

2.7 Operation Breakdown

With all modules introduced, this section presents the overall workflow of **VIPSTER**. The engine is provided with **RISC-U** machine code as input. Before

even executing the code, the data segment and arguments are bootstrapped onto the trace. Then the pseudocode, displayed in Figure 4, is followed to symbolically execute the program. First, the **program counter** and **trace counter** is initialized. VIPSTER tracks the total number of symbolic values generated for a path with a counter **#sym**.

Program Execution and Trace Generation

Instructions are executed until an exit call is reached (3-14). New instructions are fetched (4), executed (11) and their effect on the state is recorded (12). The current **trace counter** is advanced accordingly (13). If the current instruction is a **read** call, VIPSTER generates the respective symbolic intervals and increments the **#sym** counter (5, 6). For **SLTU** instructions, the engine pushes information about the available path choices on the trace and picks one path (8, 9) for further execution. This involves creating several entries on the trace and advancing the **trace counter** [9]. The current version of the engine simply always takes the **true** branch first.

When an **exit** call is finally reached, VIPSTER remembers the last entry of the forward execution - the execution brake (15). And it also prepares the **backwards trace counter** (15) for the backwards execution.

Constraint Propagation

Instructions are now fetched from the **backwards trace counter** (17), which steps through the trace backwards (26). Since propagating the constraints (23) also affects the state of the program, this effect is recorded on the trace to later undo it (24). Each witness or unsatisfiable interval gets reported to the console and the **#sym** counter is decremented (19-21). Backwards execution ends once there are no symbolic values left (16) and the **trace counter** is then set to the last entry (28) that was pushed to the trace.

Cleanup

Lastly, the trace is emptied and the state of the program reverted (37). Instructions are fetched from the **trace counter** (29,31,40), which now steps backwards through the trace (39). This phase ends once a **SLTU** is reached, that was pushed to the trace during forwards execution ($tc \leq br$, 32). If there is a path left for this instruction, it gets prepared on the trace and forward execution starts again (34). Otherwise, the instructions just get reverted and if no paths are left to explore, the cleaning reaches the start of the trace. VIPSTER then terminates the program (30).




```

1  init pc, tc, #sym
2  execute:
3  repeat
4      inst = get instruction from pc
5      if inst is read ECALL then
6          #sym++
7      end
8      else if inst is SLTU then
9          | pick path, remember possibilities on trace (tc)
10         end
11         pc = execute inst
12         record effect on trace (tc)
13         tc++
14 until inst is exit ECALL;
15 btc = tc-1, br = tc-1
16 while #sym > 0 do
17     pc = get program counter from btc
18     inst = get instruction from pc
19     if inst is read ECALL  $\vee$  inst is SLTU with unsat interval then
20         | report
21         | #sym-
22     end
23     constrain inst
24     record effect on trace (tc)
25     tc++
26     btc-
27 end
28 tc-
29 pc = get program counter from tc
30 while pc  $\geq$  0 do
31     inst = get instruction from pc
32     if inst is SLTU  $\wedge$  other path available  $\wedge$  tc  $\leq$  br then
33         | prepare next trace on trace (tc)
34         | goto execute
35     end
36     else
37         | undo inst on trace (tc)
38     end
39     tc-
40     pc = get program counter from tc
41 end

```

Figure 4: Workflow Algorithm

Example

Consider Listing 5 from the satisfiability chapter. Figure 5 shows how the program is executed on the trace with respect to the given workflow. The  represents lines 1 – 3 of the source code, where the symbolic value is generated and the variable gets incremented. Rectangle  represents the code for $(s < 10 \equiv T)$ - lines 4, 5. And  represents the case $(s < 10 \equiv F)$, hence lines 4, 6.

First (a) the code is executed forwards till the comparison is reached. **VIPSTER** then (b) chooses and executes the **true** branch till the **exit** call. The trace now contains one complete program run and backwards execution (c) is started. The engine additionally records the applied changes forwards (c). Propagation stops once the **read** call for the symbolic value is reached and no other symbolic values are left. The trace is then cleaned and the state of the program gets reverted to before the **SLTU** was executed. The second path is explored analogous but cleaning the trace is carried out till the start since no additional paths exist (e-g).

2.8 Runtime Improvement

In essence, **VIPSTER** runs through each path of a program four times - first forwards, then backwards. And finally, the cleanup process removes a chunk of at most twice the size of the original path from the trace. The first part was placed there during forward execution, the second one during the backwards propagation. Hence, **VIPSTERs** runtime for a single path is linear in the number of machine instructions that form the path.

Most state-of-the-art engines encode their path condition as boolean satisfiability problem. Checking this boolean formula is quite sophisticated [2, p.24]. A trivial approach would be to just try all possible combinations of values for the formulas variables. The set of possible combinations is exponential in the number of variables though and the path condition is even checked multiple times per path. So for a single path the runtime of an engine using a trivial solver is at least exponential in the number of symbolic values and additionally depends on how frequently the condition is checked.

So on programs with many symbolic values, **VIPSTER** should perform better compared to an engine that utilizes a trivial constraint solver.

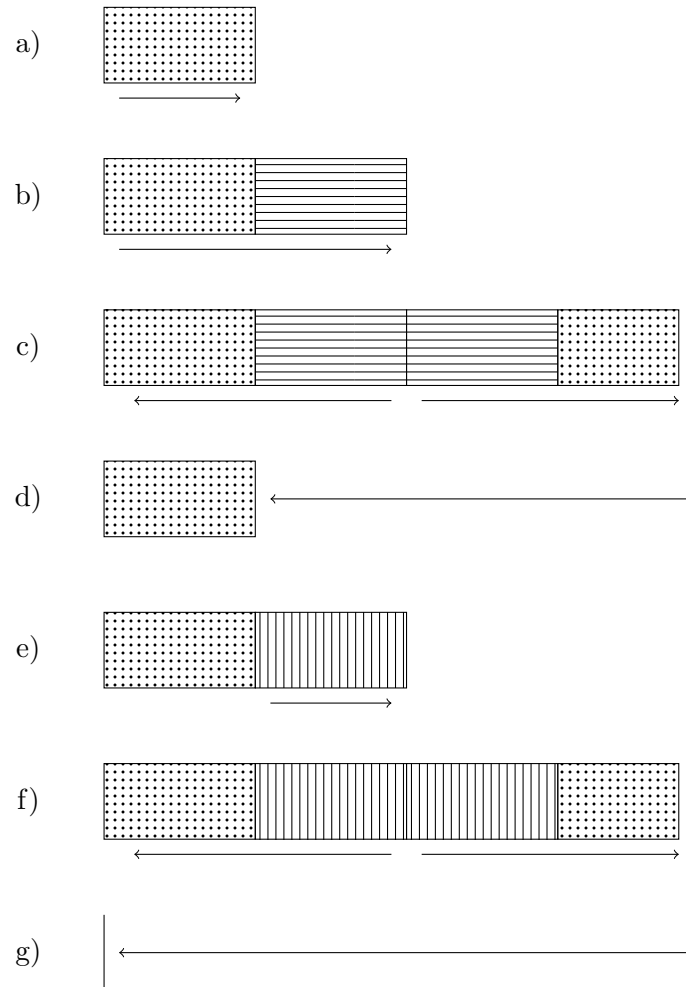


Figure 5: Workflow Visualization

3 VIPSTER in Action

This section presents several **C*** programs and how **VIPSTER** executes them. The code is usually first compiled into a **RISC-U** binary but this step is omitted here. To avoid confusion, lines of the source code are denoted by a subscript s and lines of the output by a subscript o .

3.1 Paths

Listing 9 shows a simple testfile with several possible execution paths. A character is read from the console ($2_s - 4_s$) and depending on its numerical value, the program returns either 1, 2 or 3. In detail, 1 can actually never be returned since there exists no input that can be both smaller than 15 and greater than 15 ($6_s - 8_s$). Furthermore, 2 is returned when the input is smaller than 15 and 3 is returned when the input is greater or equal to 15. Executing this code with **VIPSTER** produces the Output 10. Additional output is enabled here, so the engine provides more information about its internal workflow.

```
1 uint64_t main() {  
2     uint64_t* x;  
3     x = malloc(1);  
4     read(0, x, 1);  
5  
6     if (*x < 15) {  
7         if (*x > 15)  
8             return 1;  
9     } else  
10        return 2;  
11    return 3;  
12 }
```

Listing 9: Simple Path (Code)

To begin with, the machine code is just interpreted forwards. The **read** call (4_s) might return any character, hence the interval $[0, 255]$ is generated and saved at ***x**. **VIPSTER** internally remembers, that one symbolic value was created during forwards execution. The engine eventually comes across the first **SLTU** instruction, which was generated for the first **if** statement (6_s). The output shows that the **true** branch is picked (1_o) for exploration first. Next, the same also happens for the second **if** statement ($7_s, 2_o$). The program finally returns 1 ($8_s, 3_o$) and **VIPSTER** starts the backwards propagation. Therefore, the second **SLTU** instruction (7_s) is now encountered first, where the symbolic value is set to $[16, 255]$ to satisfy the constraint $(*x > 15)$. Then the first **SLTU** is reached and the symbolic values upper bound is moved to $[16, 14]$ to comply with $(*x < 15)$. In line 6_s , ***x** was

originally loaded from memory to perform the comparison. As this load instruction is now confined, the engine detects the invalid interval [8] and reports the unreachable path (5). Only one symbolic value was created during forwards execution, hence no values are left to verify. The trace is then cleaned up and the state of the program gets reverted to before the second SLTU instruction was executed (6). This time, the **false** branch is executed, leading to the **return** statement in line 10_s. Again the changes are propagated backwards and the interval of ***x** is first constrained to [0, 15] (7_s) and then to [0, 14] (6_s). When backwards execution reaches the **read** call, this witness is reported (10_o). Since there are again no additional symbolic values left, the propagation step is stopped here. VIPSTER reverts the state of the program but this time, there is no path left to explore at the second SLTU (7_s). Hence the state is reverted even further till the first SLTU instruction (11_o). From there, the yet unexplored **false** branch is chosen (12_o). This leads to the last possible **return** statement in line 11_s. The constraints are propagated again and the last witness is reported (15_o). When reverting the state once more, no additional paths are found anywhere on the remaining trace. As the beginning of the trace is reached, VIPSTER terminates the program (16_o).

```

1 exploring true branch from pc= 0x00010124
2 exploring true branch from pc= 0x00010138
3 tst.c exiting with exit code [1,1]
4 constraining backwards...
5 branch unreachable with [16,14]
6 undoing and resuming at pc: 0x00010134
7 exploring false branch from pc= 0x00010138
8 tst.c exiting with exit code [3,3]
9 constraining backwards...
10 satisfiable with witnesses: 0,..,14
11 undoing and resuming at pc: 0x00010120
12 exploring false branch from pc= 0x00010124
13 tst.c exiting with exit code [2,2]
14 constraining backwards...
15 satisfiable with witnesses: 15,..,255
16 undoing and terminating tst.c

```

Listing 10: Simple Path (Output)

3.2 Equality

In this Example 11, again one single character is read from the console (4_s). If its numerical value equals 10, the program will return 1 - otherwise 0 (6_s – 8_s). The comparison logic is a bit tricky here since VIPSTER only supports the SLTU instruction for comparisons. So for the expression (***x** == 10) the SELFIE compiler generates code equivalent to ((***x** - 10) < 1).

For unsigned integers, $a = b \Leftrightarrow (a - b) < 1$ holds for all $0 \leq a, b$. This is because $a = b \Leftrightarrow (a - b) = 0$, the difference between two equal values is always 0. And for any unsigned integer x , $x = 0 \Leftrightarrow x < 1$ holds. With this technique, an arithmetic underflow is created, resulting in three possible paths for the `(*x == 10)` expression [9]. The **true** branch is reachable with input 10 and the false branch with either interval $[0, 9]$ or $[11, 255]$. Hence there is one **true** and two **false** branches discovered by the engine, see 12.

```

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

Listing 11: Equality (Code)

The machine code is executed forwards first. A symbolic value $[0, 255]$ is generated through the **read** call (4_s) and it is remembered, that one such value exists throughout the program. At the **SLTU** instruction, the **true** branch is executed first (1_o) and the program returns 1 (7_s). The symbolic value is constrained to the concrete 10, which is reported as witness (4_o). Since there is no other symbolic value, the state of the program is reverted to before the **SLTU** instruction was executed (5_o). From there, the first **false** branch is executed (6_o) and constraining results in the witness $[0, 9]$. Then the trace is cleaned again and the state of the program reverted (10_s). The second **false** branch is executed (11_o) and the witness $[11, 255]$ is reported (14_o). Lastly, the state is reverted one more time but no traces are left to explore - **VIPSTER** terminates the program.

```

1 exploring true branch from pc= 0x0001012C
2 tst.c exiting with exit code [1,1]
3 constraining backwards...
4 satisfiable with witnesses: 10,..,10
5 undoing and resuming at pc: 0x00010128
6 exploring false branch from pc= 0x0001012C
7 tst.c exiting with exit code [0,0]
8 constraining backwards...
9 satisfiable with witnesses: 0,..,9
10 undoing and resuming at pc: 0x00010128
11 exploring false branch from pc= 0x0001012C
12 tst.c exiting with exit code [0,0]
13 constraining backwards...
14 satisfiable with witnesses: 11,..,255
15 undoing and terminating tst.c

```

Listing 12: Equality (Output)

3.3 Loops

The following Example 13 shows a simple loop scenario. Depending on the numerical value of the read character (7_s), the loop may be executed several times. The variable `i` serves as a counter for the iterations and is also then returned by the program. It should be noted that only basic output without any additional information is shown here 14.

```

1 uint64_t main(uint64_t argc ,
2               uint64_t*argv)
3 {
4     uint64_t * s;
5     uint64_t i;
6     i = 0;
7     s = malloc(1);
8     read(0, s, 1);
9
10    while (*s < 2) {
11        *s = *s + 1;
12        i = i + 1;
13    }
14    return i;

```

Listing 13: Finite Loop (Code)

The `read` call generates the interval $[0, 255]$ (7_s). In case the concrete input is 0, the loop is executed twice ($1_o, 2_o$). When the concrete input is 1, the loop is executed only once ($3_o, 4_o$). Finally, all values from the interval $[2, 255]$ don't meet the loop condition in the first place. Hence there are 0 iterations counted for this interval ($5_o, 6_o$).

```

1 tst.c exiting with exit code [2,2]
2 satisfiable with witnesses: 0,...,0
3 tst.c exiting with exit code [1,1]
4 satisfiable with witnesses: 1,...,1
5 tst.c exiting with exit code [0,0]
6 satisfiable with witnesses: 2,...,255
7 terminating tst.c

```

Listing 14: Loop (Output)

In this example, only one symbolic value is read and this value is initially restricted to $[0, 255]$. Since the loop counter is therefore limited to some final value and also properly incremented (10_s) each iteration, no infinite path can emerge from this loop. The next Example 15, however, shows a case where this happens. The program just reads characters from the console until the character “a” was read. Here each iteration, a new symbolic value is generated through the **read** call ($5_s, 8_s$). Hence there exists a path, where no “a” is ever entered into the console and the program keeps asking for new characters. Exploring this path eventually fills up the trace and VIPSTER terminates the program (2_o).

```

1 uint64_t main()
2 {
3     uint64_t * s;
4     s = malloc(1);
5     read(0, s, 1);
6
7     while (*s != 97) {
8         read(0, s, 1);
9     }
10
11     return 0;
12 }

```

Listing 15: Infinite Loop (Code)

```

1 executing tst.c on vipster
2 context tst.c throws uncaught tracelimit reached
3 terminating tst.c with exit code -11
4 vipster explored 1 different paths

```

Listing 16: Infinite Loop (Output)

3.4 Working with Strings

To make character handling cleaner for this example, several helper methods have been extracted from the **SELFIE** library. These are displayed in Listing

17 and provide a simple way of accessing the characters of a string. The `getChar` method is called with a pointer to a string and the position of the desired character. It then chooses the right machine word and provides the character by shifting the other characters to the left and right. Shifting is implemented via the recursive `twoToThePowerOf` method.

```

1 uint64_t twoToThePowerOf(uint64_t a) {
2     if (a < 1) return 1;
3     else return 2 * twoToThePowerOf(a-1); }
4 uint64_t leftShift(uint64_t n, uint64_t b) {
5     return n * twoToThePowerOf(b); }
6 uint64_t rightShift(uint64_t n, uint64_t b) {
7     return n / twoToThePowerOf(b); }
8 uint64_t getChar(uint64_t * from, uint64_t at) {
9     uint64_t idx; uint64_t pos;
10    idx = at / 8; pos = at % 8;
11    return rightShift(leftShift(*(from + idx), 64 -
12                          ((pos * 8) + 8)), 56);

```

Listing 17: Character Handling

In the Example 18, an eight character string is read in line $3_s, 4_s$. If the first character of that string is an “a” and the second one is “b”, the program returns 0. In case the first character is an “a” and the second one is not “b” the program returns 1. Lastly if the first character is not “a”, then 2 is returned.

```

1 uint64_t main () {
2     uint64_t * s;
3     s = malloc(8);
4     read(0, s, 8);
5
6     if (getChar(s, 0) == 'a'){
7         if (getChar(s, 1) == 'b') {
8             return 0;
9         }
10        else return 1;
11    }
12    else return 2;
13 }

```

Listing 18: Strings (Code)

For the first case, VIPSTER reports the witness $[25088, \dots, -40193]$ 19 (2_o). The binary representations of these two integers are shown in Figure 6. For both of them, the second bytes are equal and represent the character “b”. This looks like parts of the right solution, but there are some problems. The interval semantics only hold on word-level, hence for full 64 bit integers.

Character modification though is done on 8 `bit` granularity, as each character is represented with exactly 8 `bits`. In order to properly support string handling though, intervals would have to be handled on 8 `bit` level too.

```
1 tst.c exiting with exit code [0,0]
2 satisfiable with witnesses: 25088,...,-40193
3 tst.c exiting with exit code [1,1]
4 satisfiable with witnesses: 97,...,-40449
5 tst.c exiting with exit code [1,1]
6 satisfiable with witnesses: 25344,...,-159
7 tst.c exiting with exit code [2,2]
8 satisfiable with witnesses: 0,...,-160
9 tst.c exiting with exit code [2,2]
10 satisfiable with witnesses: 98,...,-1
11 terminating tst.c
```

Listing 19: Strings (Output)

All other branches show similar results. The second path, which returns 1, received the witness intervals $[97, -40449]$ and $[25344, -159]$. The tabular shows that again the second bytes properly reflect that every character until “a” and from “c” may be inserted. Therefore character “b” is properly excluded. Still, the first character should be “a” but only the lower bound of the first and the upper bound of the second interval contain “a” at the least significant bits. Lastly, the third path got the witness intervals $[0, -160]$ and $[98, -1]$. Here the character “a” is properly excluded, matching the `if` construct in the source code (6_s).

This example shows that **VIPSTER** can also handle characters, at least in some rudimental manner. Complete string support would require some more additions to the engine though, like interval semantics on byte-level.

$25088_{10} =$	0000 0000 ... 0110 0010 0000 0000 ₂
$-40193_{10} =$	1111 1111 ... 0110 0010 1111 1111 ₂
	b_{ASCII}
$97_{10} =$	0000 0000 ... 0000 0000 0110 0001 ₂
$-40449_{10} =$	1111 1111 ... 0110 0001 1111 1111 ₂
	$\leq a_{ASCII}$
$25344_{10} =$	0000 0000 ... 0110 0011 0000 0000 ₂
$-159_{10} =$	1111 1111 ... 1111 1111 0110 0001 ₂
	$\geq c_{ASCII}$
$0_{10} =$	0000 0000 ... 0000 0000 0000 0000 ₂
$-160_{10} =$	1111 1111 ... 1111 1111 0110 0000 ₂
	$\leq '_{ASCII}$
$98_{10} =$	0000 0000 ... 0000 0000 0110 0010 ₂
$-1_{10} =$	1111 1111 ... 1111 1111 1111 1111 ₂
	$\geq b_{ASCII}$

Figure 6: Strings (Binary Representations)

4 Conclusion and Related Work

The **SELFIE** system, a platform for teaching compilers, operating systems and virtual machine monitors, has been extended with a simple symbolic execution engine. The **VIPSTER** engine features a lightweight design that differs from most other state-of-the-art solutions. First of all, the symbolic values are represented through intervals, which allows to apply arithmetic operations and constraints on them directly. **VIPSTER** also does not feature a constraint solver like many other modern tools, which allows for a faster reasoning process. The engine is thereby restricted to a subset of all possible executable programs though. Several examples showed that nevertheless non-trivial code can be symbolically executed.

The Computational Systems Group is also working on a symbolic engine for **SELFIE** that only executes the machine code once without backwards propagation, as well as a version that handles intervals on byte level.

5 Appendix

References

- [1] Selfie. 2015-2018. `selfie.cs.uni-salzburg.at`.
- [2] Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.
- [3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM/SIGSAC CCS Conference on Computer and Communications Security*, 2006.
- [4] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2), February 2013.
- [5] Daniel Dunbar, Cristian Cadar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *USENIX Symposium on Operating Systems Design and Implementation*, 2008.
- [6] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.
- [7] Christoph M. Kirsch. Selfie and the Basics. *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!’17)*, 2017.
- [8] Sara Seidl. Symbolic execution with SELFIE: Logics, 2018.
- [9] Manuel Widmoser. Symbolic execution with SELFIE: Arithmetics, 2018.

List of Figures

1	Symbolic Expression Tree	8
2	Simple Path Tree (Visualization)	11
3	Strings in VIPSTER: Trace	18
4	Workflow Algorithm	22
5	Workflow Visualization	24
6	Strings (Binary Representations)	32

List of Listings

1	ADD with symbolic trees	8
2	ADD with intervals	10
3	Simple Path Tree (Code)	10
4	Simple Aliasing Example	16
5	Simple Backwards Example	17
6	Strings in VIPSTER: Code	18
7	Strings in VIPSTER: Assembly	18
8	Generating Symbolic Values	20
9	Simple Path (Code)	25
10	Simple Path (Output)	26
11	Equality (Code)	27
12	Equality (Output)	28
13	Finite Loop (Code)	28
14	Loop (Output)	29
15	Infinite Loop (Code)	29
16	Infinite Loop (Output)	29
17	Character Handling	30
18	Strings (Code)	30
19	Strings (Output)	31