Department of Computer Sciences
University of Salzburg

Bachelor Thesis

Symbolic Execution with Selfie Logics

Salzburg, 2019

Sara Seidl
sara.seidl@stud.sbg.ac.at

Academic Supervisor:
Professor Christoph Kirsch

Correspondence to:
Universität Salzburg
Fachbereich Computerwissenschaften
Jakob–Haringer–Straße 2
A–5020 Salzburg
Austria

Contents

List of Figures

Abstract

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 solver we developed is based on interval modulo arithmetic rather than bit-vector logics used by many other symbolic execution engines. Further, constraint solving is not achieved by solving formulas but entirely through execution of code. It is interval modulo arithmetic in combination with this constraint solving technique that allows us to solve constraints fast. The solver is complete as long as the set of concrete values a symbolic value represents can be expressed as an interval.

The challenge is to be able to detect if the 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 Introduction

In this thesis we present Vipster, a symbolic execution engine developed in course of our bachelor project. Vipster symbolically executes binary code and analyzes paths that exit with an error. It reasons about the satisfiability of the path conditions collected for these paths and generates concrete witnesses. It does so without using computationally expensive constraint solvers. Instead, it employs a technique we refer to as *Backward Execution*. This technique enables Vipster to solve constraints fast. Every path is executed at most twice, once forward to collect the path condition and once backward to evaluate it. Backward Execution is only complete for a subset of all possible programs, but the solver is able to detect if Backward Execution is incomplete.

1.1 Symbolic Execution

Concrete execution is performed on concrete input values. Statements may change these values, but they always remain concrete. When execution is finished a single path of execution has been followed. Symbolic execution [1], [2] is not restricted to concrete values. In symbolic execution input values are allowed to be symbolic. A symbolic value represents the set of values the input may be taken from. Statements modify the set represented by a symbolic value. Whereas arithmetic statements solely modify the set of values, conditional control flow statements have a more sophisticated impact on symbolic values. The execution of a conditional control flow statement that involves a symbolic value and thereby a set of values will in the simple case follow either the true or the false branch. The interesting case is when both paths are a valid option to continue execution, meaning, a subset of values satisfies the true branch while another subset satisfies the false branch. At this point execution is forked and both paths are explored. As a result, when symbolic execution is finished all possible paths have been explored.

1.1.1 Execution State

The execution state of a program can be described in terms of the values of all program variables and the program counter indicating the next instruction that will be executed. Symbolic execution considers the execution state of individual paths. The symbolic state of a path additionally consists of a path condition and a symbolic mapping. The path condition is often a first-order Boolean formula over symbolic expressions that is collected during symbolic execution. At branching points at which execution is forked the path condition is updated by adding the branching condition for the taken path to it. At any point during execution the path condition describes all conditions that must be satisfied in order to reach that point of execution. The symbolic mapping maps variables to symbolic expressions or values.

Many different engines for symbolic execution have been developed [3], [4]. The techniques used by these engines, can be divided into three main categories: online symbolic execution, offline symbolic execution, also referred to as concolic testing [5] and hybrid symbolic execution as introduced by Mayhem [6].

1.2 Overview

I will briefly summarize my work on Vipster and present my contributions. This section also provides an overview of the content and structure of my thesis.

The Vipster system has three main components, the execution engine, the constraint solver and the traversal engine. Most of my time went into the development of the constraint solver.

Prerequisites - Data Structure Before any of Vipster's components were build I implemented a trace data structure that can store enough information about execution to enable us to perform symbolic execution. My first implementation is very similar to the trace that it is now used by Vipster. Next, the symbolic execution engine was developed to operate on and with the trace.

Forward Execution - Collecting Constraints The symbolic execution engine developed by my colleague Manuel Widmoser executes a single path at a time using the *symbolic forward semantics* of instructions. The engine stores information for every instruction it executes on the trace. It also stores branching points at which execution is forked.

The execution engine already constrains symbolic values during forward execution and stores these constraints on the trace as well. However, the engine ignores issues that arise in course of forward execution: *Register Memory Gap* and *dependencies amongst variables.* As a consequence, the execution engine also reports false positives.

Backward Execution - Solving Constraints To address the problem of false positives, I developed an algorithm that solves the mentioned issues *after* the forward execution of a single path is finished and a trace of information and constraints has been collected. Aside from reasoning about the *reachability* of the reported error the constraint solving algorithm also generates a *concrete witness* for the path that was explored when the error occurred.

The constraint solver I developed executes the explored path backwards and stores information about its backward execution forward on the trace. *Backward Execution* propagates constraints collected during forward execution upwards the explored path and reflects them onto the initial interval using the *symbolic inverse semantics* of instructions. I defined the symbolic inverse semantics of instructions so that:

- arithmetic instructions *propagate constraints* upwards the explored path
- memory instructions *constrain* symbolic values in memory

The inverse semantics of memory instructions is key in bridging the Register Memory Gap and resolving dependencies amongst variables. Backward Execution is only complete for a subset of all possible programs, but the solver is able to detect if it is still complete.

Path Exploration The traversal engine I implemented is what allows us to explore all feasible paths. It uses a simple depth-first search algorithm that rewinds backward and forward execution to a previous branching point from which another path can be followed. It does so by walking backwards the trace to revert all changes.

1.3 Contribution

We present Vipster, a self-contained symbolic execution engine and part of the selfie project [7]. What distinguishes Vipster from other symbolic execution engines is its symbolic execution technique and the constraint solving approach that allows Vipster to solve constraints fast. Vipster's execution engine employs a technique that combines concepts used in online and offline symbolic execution.

Offline Symbolic Execution Vipster reasons about a single execution path at a time and maintains a trace. The execution technique used is a combination of concrete and symbolic execution.

Online Symbolic Execution Although Vipster executes each path separately it never has to re-execute a previous instruction due to a backtracking strategy. At each branching point Vipster stores all feasible branches with as little strain on memory as possible in our setting.

Vipster's symbolic execution is driven by concrete execution, as are different concolic testing approaches [3]. But other than these approaches Vipster does not distinguish between symbolic and concrete execution but combines them. The fact that Vipster models symbolic values as intervals allows us to define a concrete execution semantics on symbolic values.

Vipster generates concrete test inputs for execution paths that exit with an error without using an SMT solver. Instead, Vipster uses its own constraint solver that employs a strategy that allows it to solve constraints fast.

Symbolic execution engines that perform bit-precise reasoning often use bit-vector logics and hence they rely on SMT solvers for the quantifier-free theory of fixed-size bit-vectors [8]. The path condition generated by these symbolic execution engines is encoded as an SMT formula. The current state-of-the-art procedure used by the SMT solvers to determine satisfiability of such a formula is bit-blasting, a flattening technique to translate an SMT formula into a SAT formula. Translating and deciding the satisfiability of SMT formulas is still a bottleneck in symbolic execution.

Vipster's 64-bit-precise reasoning in contrast is based on interval modulo arithmetic. The path condition generated by Vipster's execution engine is not a formula but implicitly represented in control flow and the value of symbolic intervals. As a consequence, the constraint solver reasons about the reachability of a given path, rather than the satisfiability of a formula. Executing code once forward and once backward suffices to determine the reachability of a path and generate concrete witnesses for it.

The design of Vipster is based on two basic principles we hoped to fulfill:

- **Performance**
  Constraint solving is still a bottleneck in symbolic execution. Vipster employs a constraint solving technique that trades completeness to increase performance. We call this technique *Backward Execution.*
- **Simplicity**
  Because Vipster is part of the selfie project a major objective when designing the system was simplicity. The challenge we were presented with was finding the minimal setup to perform symbolic execution.

1.4 Collaboration

This thesis is based on joint work with Simon Bauer and Manuel Widmoser with whom VIPSTER was designed and engineered over the past year. This thesis is the second in a three-part-series, each focusing on a different part of our joint work.

A detailed introduction to symbolic execution, followed by an overview of other engines and a comparison to ours can be found in the thesis of Simon Bauer. He also presents and analyzes experiments in his work. Symbolic forward execution is the topic of Manuel Widmosers thesis. In his work he discusses how the symbolic execution engine operates. Because VIPSTER represents symbolic values as intervals, modulo interval arithmetic [9], [10] is one main topic in his thesis. He also introduces the trace, the data structure VIPSTER uses to collect constraints during forward execution (and more). In my work I focus on solving the collected constraints and on exploring all possible execution paths. However, in order to do this properly I found it necessary to provide some insight into forward execution my thesis as well.

1.5 Outline

This thesis is divided into two sections. The first section gives a brief overview or our VIPSTER system. It describes how VIPSTER models symbolic state and introduces the main components of VIPSTER. This section is concluded by illustrating the constraint solving algorithm that VIPSTER uses.

The second section discusses the implementation of VIPSTER's constraint solver in greater detail. At the beginning the underlying data structure VIPSTER operates on is presented. The focus of this section is then on constraint solving and path exploration.

2 Vipster Design

This section provides a high-level introduction of the Vipster system. It first presents the execution engine that symbolically executes a `RISC-U` binary generated by the selfie compiler. `RISC-U` is a tiny subset of the `RISC-V` instruction set featured in selfie. A list of the 14 `RISC-U` instructions can be found in Section 3.2.1. Next, this section introduces the constraint solver that reasons about satisfiability of each path explored by the engine and the traversal engine that enables exploration of different paths. This section is concluded by illustrating the constraint-solving algorithm employed by the constraint solver.

2.1 Modeling Execution State

To illustrate how the components of the Vipster system operate we will first explain how Vipster models execution state. As Vipster symbolically executes `RISC-U` binaries we speak about execution state in terms of the values of registers and memory locations rather than the value of variables. An important element in modeling execution state is the trace, which will be also introduced.

2.1.1 Concrete and Symbolic Values

Vipster represents symbolic values as well as concrete values as clockwise intervals over a finite set of natural numbers. The terms value, symbolic value and interval are used interchangeably throughout this thesis.

Definition 1 (Clockwise Interval)**.** *Let $\mathbb{N}_{MAX}$ denote the finite set of natural numbers $\mathbb{N}_0$ modulo $MAX \in \mathbb{N}_0$. Let a and b be two integers modulo MAX. A clockwise interval [9] $[a,b]_{MAX}$ with a lower bound a and upper bound b is a set of natural numbers x given by*

$$[a,b]_{MAX} = \begin{cases} \{a, a+1, ..., b\}, & a \le b \\ \{a, a+1, ..., MAX-1\} \cup \{0, 1, ..., b\}, & a > b \end{cases}$$

Throughout this thesis $[a,b]$ refers to $[a,b]_{MAX}$.

Definition 2 (Inclusion)**.** *The inclusion of intervals $[a',b'] \subset [a,b]$ is defined as follows:*

$$[a',b'] \subset [a,b] \iff (a' \neq a \lor b' \neq b) \land \begin{cases} a \le a' \land b \ge b', & a' \le b' \land a \le b \\ a \ge b' \lor b \le a', & a' \le b' \land a > b \\ a \ge a' \lor b \le b', & a' > b' \land a > b \\ [a,b] == \{0, ..., MAX-1\}, & a' > b' \land a \le b \end{cases}$$

Definition 3 (Union). *The union of two intervals $[a, b] \cup [c, d]$ is not closed over clockwise intervals in general. However, the result of the union operation is again an interval iff one of the following cases holds:*

- *Case 1 (No wrap-around):* $(a \leq b \wedge c \leq d) \wedge$
  - (a) $((a-1) \leq d \wedge c \leq (b+1))$
  - (b) *w.l.o.g.* $b == (MAX-1) \wedge c == 0$
- *Case 2 (Single wrap-around w.l.o.g.):* $(a > b \wedge c \leq d) \wedge$
  - (a) $(b+1) \geq c$
  - (b) $(a-1) \leq d$
  - (c) $(b+1) \geq c \wedge (a-1) \leq d$
- *Case 3 (Double wrap-around):* $(a > b \wedge c > d) \wedge$
  - (a) $((c \leq b+1) \vee (d \geq (a-1)))$
  - (b) *otherwise*

Then the union of two intervals is then defined as follows:

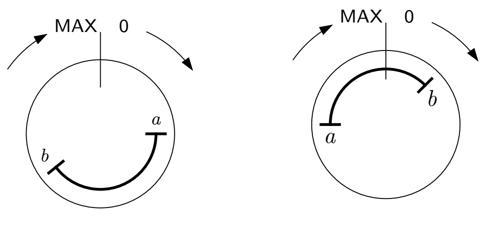
$$[a, b] \cup [c, d] = \begin{cases} [min(a, c), max(b, d)], & \textit{case 1.a, case 3.b} \\ [a, d], & \textit{case 1.b} \\ [a, max(b, d)], & \textit{case 2.a} \\ [min(a, c), b], & \textit{case 2.b} \\ [0, MAX], & \textit{case 2.c, case 3.a} \end{cases}$$

Definition 4. *Terminology used to describe intervals:*

- *We refer to the case $a > b$ as wrapped interval.*
- *The case $a == b$ describes a concrete interval representing a single concrete value.*
- *When the bounds of an interval $[a, b]$ are modified such that an interval $[a', b']$ with $[a', b'] \subset [a, b]$ results, the interval $[a, b]$ is said to be constrained to $[a', b']$.*

Corollary 1. *A wrapped interval $[a, b]$ is equivalent to the union $[a, MAX-1]$ and $[0, b]$.*

The advantage of using intervals to represent symbolic values is the compact representation and well-defined interval arithmetic. However, it is also interval arithmetic that puts limitations on our engine and solver in what they can reason about. Restrictions on completeness result from those limitations. But since the limitations are known, the engine can detect if it becomes incomplete during execution.



(a) Clockwise Interval (b) Wrapped Interval

Figure 1: In modulo arithmetic numbers are placed at equal intervals on the circumference of a circle, instead of along a straight line.

2.1.2 Execution State and Intervalls

The symbolic execution state of VIPSTER's execution engine is described by the values of registers and memory locations. Since register and memory values are represented by intervals and hence sets of values, a symbolic execution state X represents a set of concrete execution states $x_1, x_2, ..x_n$.

Executing an instruction takes the engine from a symbolic state X to a symbolic state Y. Aside from the value of the program counter, state X and state Y differ in at most a single register or memory value. Executing a sequence of instructions takes the machine from an initial symbolic state A to a final symbolic state B.

Constraining Constraining an interval $[a, b]$ in state X to an interval $[a', b'] \subset [a, b]$ constrains state X to state $X' = (X \backslash [a, b]) \cup [a', b']$. Constraining a state shrinks the set of concrete execution states represented by that state. We will represent this relation as $X' \subset_s X$ (Note that this is not the standard subset definition).

Splitting Splitting an interval $[a, b]$ into two disjoint intervals $[a_1, b_1], [a_2, b_2] \subset [a, b]$, with $[a_1, b_1] \cup [a_2, b_2] = [a, b]$ forks execution state X. Two disjoint execution states $X_1 = (X \backslash [a, b]) \cup [a_1, b_1]$ and $X_2 = (X \backslash [a, b]) \cup [a_2, b_2]$ result. Note that X_1 and X_2 are constrained states.

This provides insight into the connection between the symbolic state of our engine and the concrete execution states it represents. Throughout this thesis we will describe symbolic execution mainly in terms of intervals that change rather than the set of states that gets modified.

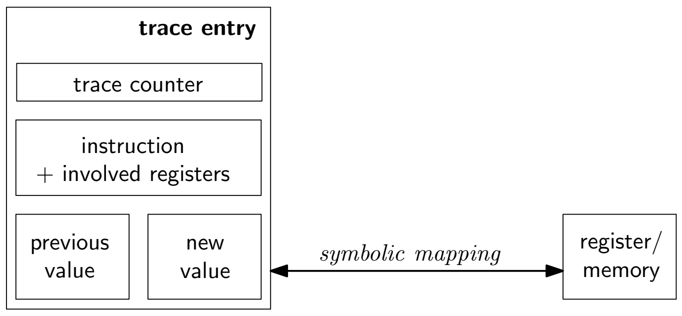


Figure 2: Mapping between register or memory location and (abstract) trace entry

2.1.3 Trace

The trace is the core component of the VIPSTER system. Every other component of the system was built on top of the trace and therefore heavily depends on it. For every instruction executed by the execution engine information is stored on the trace in form of a *trace entry*.

Trace entry The trace entry, as shown in Figure 2, is a record that shows what operation was performed and which register or memory location was modified. More precicely it remembers the value present in register or memory before the modification and also the new value resulting from execution of the instruction. Every trace entry has a unique identifier by which it can be located on the trace referred to as the *trace counter*.

There are already entries on the trace when symbolic execution starts. They represent stack arguments, values in the data segment and initial register values as intervals and were put there by VIPSTER during the initialization step. Hence every value ever used in a computation or any value resulting from a computation is represented by an entry on the trace.

A description of the trace data structure is provided in Section 3.1. For more details on building the trace during forward execution, see Manuel Widmoser's thesis.

2.1.4 Introducing Symbolic Values

The only way to introduce symbolic values in a program is through the `read` system call. Depending on the number of bytes to read, one or more intervals are initialized with maximal cardinality and are stored on the trace.

2.1.5 Symbolic Mapping

Registers and memory locations can not hold intervals directly, so they have to be stored elsewhere. VIPSTER then maps every register and every memory location to such an interval. Since the trace stores the values of register and memory locations when they change we use the trace and its unique identifiers to create the mapping. Every register or memory location stores the trace counter of the trace entry that represents the most recent change. This is illustrated in Figure 2.

2.1.6 Path Condition

VIPSTER's path condition is implicitly represented by control flow information stored on the trace and also by the values on the trace. The sequence of instructions recorded during execution represents the currently followed path from which the path condition could be derived. Additionally, at any point during execution the values on the trace also reflect the conditions that must be satisfied up to that point. Constrained values are of special interest. Manuel Widmoser explains in his thesis how this is achieved and why this is the case. VIPSTER reads the path condition from the values on the trace.

2.2 Components of Vipster

The three main components of the VIPSTER system are the execution engine, the constraint solver, and the traversal engine.

The execution engine symbolically executes a single path at a time and accumulates an implicit path condition on the trace. If the executed path runs into an error the constraint solver evaluates the path condition. To explore further paths the traversal engine uses a backtracking algorithm to revert to a previous execution state from which another path can be followed.

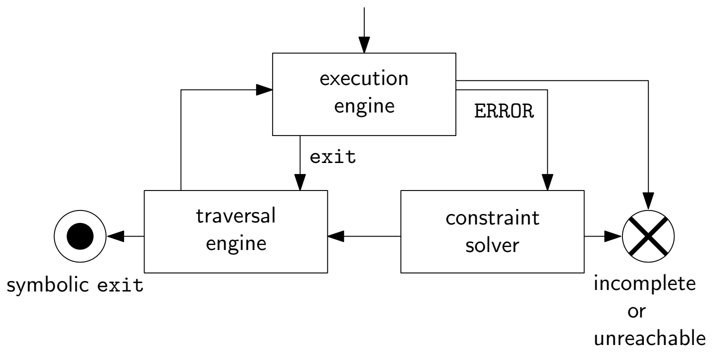


Figure 3: The three main components of VIPSTER.

2.2.1 Execute

VIPSTER's execution engine symbolically executes `RISC-U` code by interpreting `RISC-U` instructions. To operate on symbolic values, the *symbolic forward semantics* of each instruction has to be defined. Using intervals to represent symbolic values allows us to base the semantics on modulo interval arithmetic. During forward execution each instruction is then interpreted according to its symbolic semantics and the path condition is accumulated on the trace. A path that exits with an error has to be evaluated by VIPSTER's constraint solver.

The key instruction during forward execution is `SLTU`. This instruction is the cause for branching. I recommend reading about this instruction in the thesis of Manuel Widmoser. Roughly speaking, `SLUT` compares two intervals and takes the engine from state X to state Y by computing the result of the comparison. But before the instruction is executed the engine might split one of the compared intervals and thereby forks the execution state X. This then causes the currently executed path to diverge into two paths each continuing from one of the forked states. SLTU is executed after the state is forked. The example below shows an interval $[0, 256] \in X$ that is split before of execution an SLTU instruction.

$$[0, 256] < [33, 33] = \begin{cases} [0, 32] \in X_1 \subset_s X & \textit{true} \text{ state} \\ [33, 256] \in X_2 \subset_s X & \textit{false} \text{ state} \end{cases}$$

If the path diverges both the false and the true branch are recorded on the trace. Each branch gets associated with the corresponding constrained interval and execution continues, exploring the true branch first. SLTU is the only instruction that can constrain symbolic values during forward execution.

What is important to understand is that the execution engine really executes instructions. Intervals are modified in course of execution and a decision, more precisely branching, is based on their current value. This means that the execution engine only explores a path if that path is reachable according to the engine's current knowledge. However, the execution engine is somehow sloppy or lazy and might follow a path that seems to be reachable, even when it is not. The two reasons for this behavior are *dependent variables* and what we call the *Register-Memory-Gap*. To keep the execution engine simple, we decided not to address these issues during forward execution. We will later see that these issues can be resolved during the constraint solving process more easily.

As mentioned, details on forward execution and modulo interval arithmetic are discussed in the thesis of Manuel Widmoser.

2.2.2 Constrain

VIPSTER's constraint solver evaluates the trace recorded by the execution engine for every path that exits with an error. This evaluation has two objectives:

- check reachability for the given path
- generate concrete witnesses for the given path

The first objective addresses the issue of false positives, errors reported by the execution engine even though the point of execution at which they occurred is not reachable. Reasoning about reachability requires bridging the Register-Memory-Gap and resolving dependencies amongst variables.

Developing VIPSTER we only had this first objective in mind and attempted to design an algorithm that can perform the reachability check efficiently. Given a trace of execution for a path the algorithm should determine whether the path is reachable or not. The algorithm we developed executes the explored path *backwards* to solve the issue of Register-Memory-Gap and dependent variables. In the process it also generates concrete witnesses for a path that is reachable.

Backward Execution, as it turned out, connects both objectives and we shift our focus from reachability check to witness generation. A path is reachable if and only if there is a satisfiable assignment for the initial symbolic values. This satisfiable assignment is further a witness for the path. In the attempt to find a satisfiable assignment the constraint solver also reasons about reachability.

2.2.3 Explore

VIPSTER explores execution paths using a depth-first search algorithm. The execution engine explores a path until it exits before turning over control to the constraint solver or the traversal engine. VIPSTER's traversal engine then uses a backtracking algorithm to rewind execution and execution state to the previous branching instruction. First the effects of the constraining process are reversed. Then forward execution is rewinded up to the branching instruction. Last the traversal engine modifies the restored state and turns over control to the execution engine, closing the circle.

The modifications performed by the traversal engine in the last step will cause the execution engine to follow the next path. If there are no paths left to explore, the traversal engine rolls back the entire forward execution and exits.

2.3 The Constraint-Solving Algorithm

"Given a trace of execution, is there a satisfiable assignment of symbolic values, such that execution would produce this trace, and if so, what does this assignment look like?"

Vipster's constraint solver attempts to answer both questions by symbolically executing the explored path backwards. We will first explain the idea behind witness generation and *Backward Execution* and later answer the question of reachability.

2.3.1 Witness Generation

Vipster's constraint solver tries to generate concrete input values that cause execution to follow the traced path. The intuition behind our approach is illustrated in Figure 4.

Arithmetic statements executed during symbolic execution *modify* intervals. Their execution does not impose any conditions on the initial symbolic input values. The same holds for conditional control flow statements that evaluates to either true or false for the whole interval. In these cases a single path is followed.

Conditional control-flow statements whose result depends on the subset of the interval considered cause execution to fork. At this point Vipster *constrains* the involved interval. The interval is split, one subset satisfying the true branch the other satisfying the false branch. Vipster continues execution along one possible path using the corresponding constrained interval. Additionally, the other constrained interval is remembered on the trace to enable exploring that path at a later point. Constraining an interval does affect the initial interval.

Backward execution propagates the effect of constraining an interval during execution upwards the explored path. The constraining process is finished when all constraints are reflected by the symbolic input values.

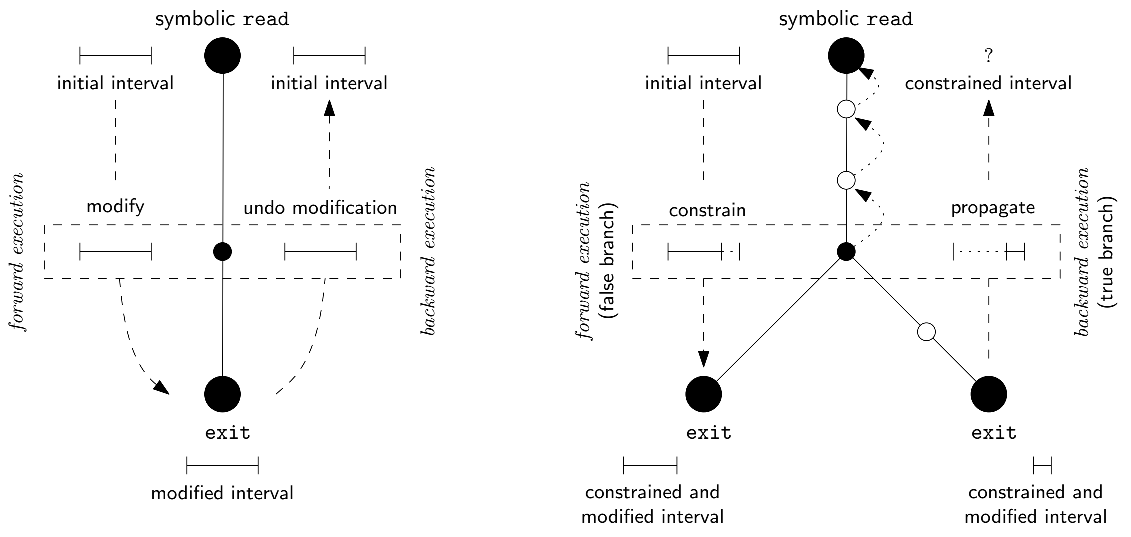


Figure 4: An illustration of the effect of arithmetic instructions (left) and conditional instruction (right) on the initial interval.

2.3.2 Backward Execution

Backward Execution is explained using the concept of modifying execution state introduced in the previous section.

Forward execution of a sequence of instructions takes the machine from an initial state A and no constrained values to a final state B. During execution values might have been constrained by SLTU. Backward Execution starts in state B and executes the sequence of instruction *in reverse* attempting to reach a distinct state $A' \subseteq_s A$.

Executing an instruction in reverse we start in a possibly constrained state $Y' \subseteq_s Y$. In this state the instruction that was responsible for the change of state is known. The goal is to derive a state $X' \subseteq_s X$ using the *inverse semantics* of an instruction. State X' is the state that was observed before the instruction was executed. In this process one of three things can happen:

- a distinct state $X' \subseteq_s X$ can be derived such that all other states X'' that could be derived are contained within X', hence $X'' \subset_s X'$
- several disjoint states $X'_1, X'_2, ... \subset_s X$ can be derived
- no state X' can be derived

If no state X' can be derived, it means no state X' exists such that after the execution of the instruction state Y' can be observed. As a result this point in execution and therefore the final state B is not reachable as the path condition is unsatisfiable. In case VIPSTER constraint solver can derive a distinct state X' backward execution is continued with the next instruction. When the second case occurs the solver aborts. At this point branching is necessary to stay complete since a set of states $X'_1, X'_2, ... \subset_s X$ could potentially result in several initial states $A'_1, A'_2, ... \subset_s A$.

Branching during backward execution is beyond the abilities of our simple solver. It is, however, always possible to detect this case.

If Backward Execution reaches a state $A' \subseteq_s A$ the symbolic values in state A' represent witnesses for the reachability of state B.

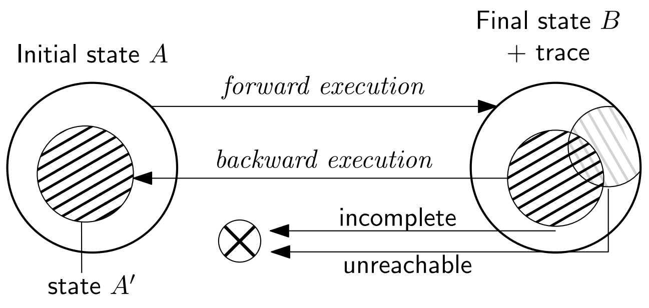


Figure 5: The concept of *Backward Execution*.

3 Vipster Constraint Solver

Before discussing the specifics of Vipster's constraint solver and the traversal engine in this section, the data structure they depend on, the trace, is presented in greater detail.

3.1 Trace Data Structure

The trace is an array of *trace entries* which are comprised of a *program counter*, a *trace counter*, an *upper and lower bound* and a *state flag*. For each executed instruction a trace entry is created.

Program Counter The program counter identifies the instruction and the involved registers for which the entry was created. A sequence of program counter values stored on the trace represent the currently explored path.

Trace Counter In general the trace counter uniquely identifies a trace entry and is used for the mapping from registers and memory locations to intervals on the trace.

As part of the trace entry it refers to the interval stored in register or memory before the instruction modifies that register. This interval is represented by the trace entry that the register or memory is mapped to before the instruction is executed. Note that this trace entry again has a trace counter pointing to a previous value. Following this singly-linked list of trace counters we obtain the history of that register of memory location.

Bounds Executing the instruction might change a register or memory value. Upper and lower bound represent the new interval that results from executing the instruction. This interval can be the result of an arithmetic instruction or a comparisons instruction, but also the interval loaded into a register or stored into memory. After executing the instruction and putting the bounds on the trace, the register or memory location is updated and maps to the trace counter of the newly created trace entry.

State Flag The state flag is used to track the state of an interval during forward and backward execution. States are `SYMBOLIC`, `CONSTRAINED`, `UNSATISFIABLE` or `CONCRETE`. Transitioning between these states is only possible in one direction:

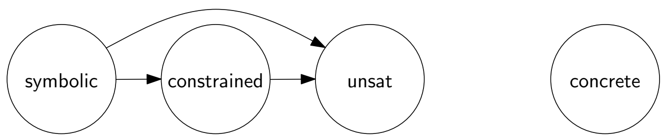


Figure 6: State transitions of symbolic values

The intervals that are put on the trace as result of the read call are marked as `SYMBOLIC`. A symbolic interval can become `CONSTRAINED`, either by `SLTU` during forward execution or a `LOAD/STORE` during backward execution. Both symbolic and constrained intervals can become `UNSATISFIABLE` during backward execution. Constant intervals that never were symbolic are `CONCRETE`. The state of a computed value is derived from the state of the involved operands. To see how state is traced during forward execution I recommend reading about this in the thesis of Manuel Widmoser.

3.1.1 Using the Trace

The execution engine builds the trace. It collects information for every instruction that it executes in form of trace entries. By looking at the whole trace rather than on individual trace entries we can gain even more knowledge. The trace during forward execution represents at any point

- the currently explored execution path
- the path condition collected up to this point, represented by (constrained) values
- the state of every register and memory location
- every path branching from the current execution path that has not yet been explored
- a history for every register and memory location ever modified during execution

The constraint solver uses the trace to propagate constraints by executing the path backwards. It does not modify the original trace but extends it in course of the constraining process. This in turn allows the traversal engine to reverse the constraining process. Figure 7 explains the general usage pattern. The trace after successful Backward Execution represents

- the applied path condition (propagated constraints) on the execution path
- a set of concrete input values that lead execution down this path

The traversal engine uses the trace to restore a previous execution state. It walks backwards on the trace reading each trace entry to reverse the modification that the entry represents. This is done by restoring the previous value the trace counter points to. The traversal engine stops at the most recent branching point (`SLUT` instructions) that has an unexplored path left. Figure 7 also illustrates this process.

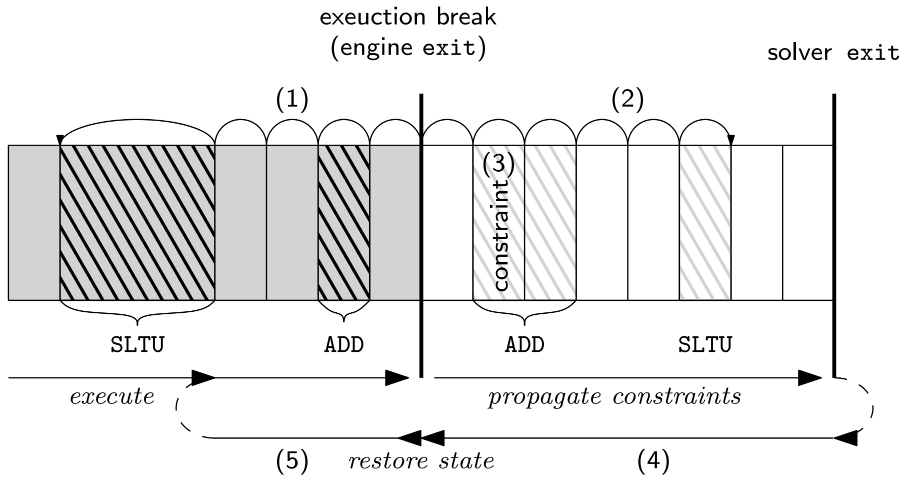


Figure 7: The solver reads the sequence of program counters in reverse order (1) and executes the instruction with its inverse semantics. For every instruction executed it stores information on the trace (2) just like the execution engine. If an instruction (here `ADD`) causes an interval to be constrained an additional trace entry will created (3). This entry represents the applied constraint and remembers the previous unconstrained value (trace counter).
The traversal engine walks the trace backwards (4, 5) up to the most recent `SLTU`. For every entry it restores the previous value thereby undoing the modification. For entries created by the solver (4) this reverses the constraining process whereas for entries created by the engine (3) it rewinds execution.

3.2 Backward Execution - Constraint Solving

Vipster solves constraints recorded on the trace by executing the recorded path backwards. Backward execution propagates the constraints back an reflects them onto the initial interval. Each instruction has a corresponding *symbolic inverse semantics* used for backward execution. This section starts with a general introduction to inverse semantics, followed by an explanation of the inverse semantics of arithmetic instructions. It then discusses the inverse semantics of control-flow instructions before a detailed analysis of memory instructions concludes this section.

3.2.1 Instructions and Inverse Semantics

Selfie features a tiny subset of the `RISC-V` instruction set that is called `RISC-U`. 14 `RISC-V` instructions suffice to support all of the C* language features. These are the load-immediate instruction `LUI`, the arithmetic instructions `ADDI`, `ADD`, `SUB`, `MUL`, `DIVU`, `REMU` and `SLTU`, the control-flow instructions `JAL`, `JALR` and `BEQ`, the memory instructions `SD` and `LD` and the `ECALL` instruction used for system calls. The forward semantics of each instruction is defined and analyzed in the thesis of Manuel Widmoser. But since the inverse semantics of an instruction is derived from its forward semantics and strong assumptions are made about how the trace is prepared during forward execution, the forward semantics will briefly be outlined.

3.2.2 Annotations and Illustration

To stay consistent in analyzing the inverse semantics of instructions simple annotations are introduced. These annotations represent different steps that are performed when an instruction is interpreted. The '**?**' after annotations represents optional steps.

- **@prepare** Before an instruction can be interpreted it might be necessary to prepare the trace and/or the engine. This annotation combines every action that is performed in advance to enable correct interpretation of an instruction.
- **@save** Before an instruction is executed information is stored on the trace. The program counter is the one piece of information that is stored for every instruction. Some instructions may additionally store the content of the to be modified register or memory location.
- **@execute** This annotation refers to the semantics of an instruction during forward execution. Executing an instruction that modifies a register or memory location results in a new interval value on the trace.
- **@restore** This annotation refers to the semantics of an instruction during backward execution. It describes the process of reversing the effects of forward execution on the modified register or memory location. This step only reads the @save value previously stored on the trace.
- **@constrain** This annotation also refers to the semantics of an instruction during backward execution. Other then @restore, the @constrain step modifies registers or read memory locations not by reading a previous value but by computing a new constrained value. This is where the inverse semantics of the instruction is used.
- **@update** After the interpretation of an instructions a new trace entry has been created. This annotation refers to actions performed to update the state of the engine before the next instruction can be executed.

Figure 8 illustrates the relation between the annotations and a trace entry. In course of each step different parts of the trace entry are created. Figure 9 shows how the annotations can be used to describe the execution of an instruction.

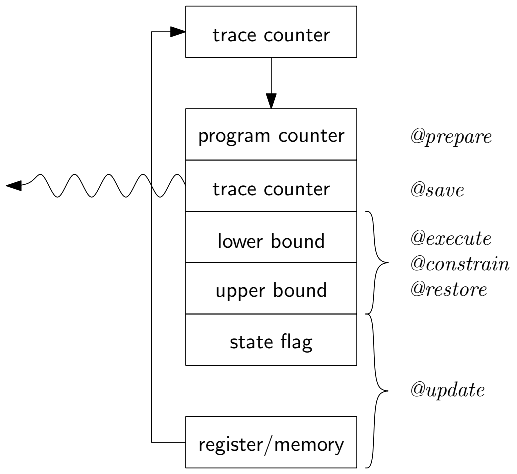


Figure 8: During @prepare the trace counter identifying the instruction is stored on the trace. @save stores the trace counter of the value currently in the to be modified register or memory location. Lower and upper bound are computed or restored during @execute, @constrain or @restore. In the @update step the state of the newly computed value is determined and the register or memory location that was modified is mapped to the created trace entry.

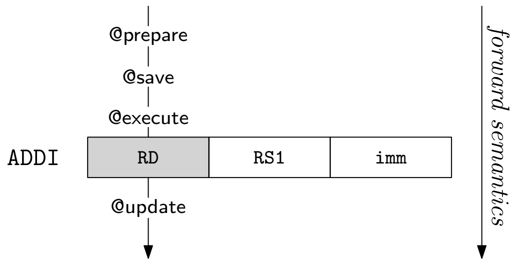
3.2.3 Inverse Semantics of Arithmetic Instructions

The inverse semantics of arithmetic instructions is defined using the forward semantics of others. For instance the effect of `ADD` is reversed by subtraction as it is defined for `SUB`. When discussing these instructions, we focus on explaining how the trace is used to propagate the constraints.

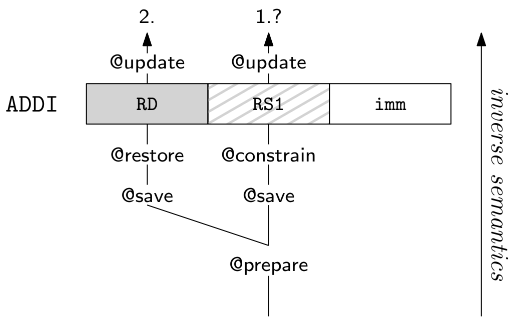
All arithmetic instructions define operations that use two source registers, namely `RS1` and `RS2`. The result of the operation is placed into a destination register `RD`. Care must be taken when a source register is also the destination register. In this case the operand value is overwritten and the initial operand value has to be retrieved from the trace using the trace counter.

Abstract Idea We are given the symbolic result of an operation and its initial operands, one of which is symbolic. The symbolic result can be constrained, meaning it differs from the result that was calculated during forward execution. In this case operands and result do not 'fit'. Since the result and the concrete operand are assumed to be correct, the symbolic operand must be wrong and has to be corrected. This correction propagates the constraint onto the source register.

Property Propagating the constraint requires finding the minimal lower and maximal upper bound for the symbolic operand such that if the instruction is interpreted forward using the new bounds the resulting interval is either equal to or contained in the constrained result. This property becomes important when we consider division, multiplication, and remainder.



(a) ADDI executed forward



(b) ADDI executed and constrained backwards

Figure 9: Figure (a) shows the steps performed during forward execution of `ADDI`. Only `RD` is modified, `RS1` is not changed. Figure (b) illustrates the backward execution of `ADDI`. If the modified register `RD` has been constrained, the constraint gets propagated onto the responsible symbolic operand (here `RS1`).

ADDI The `ADDI` instructions interpreted by the forward execution engine *adds* an immediate value to the content of `RS1` and puts the result into `RD`. Interpreted backwards `ADDI` *substracts* the immediate value from `RD` to obtain the value for `RS1`. If `RD` has been constrained in a previous step the new value obtained for `RS1` differs from its value during forward execution and `RS1` has to be constrained.

- **@prepare** The original operand value of `RS1` is retrieved and the calculation to obtain the new value for `RS1` is performed (Table 1).
- **@save @constrain @update?:** If original and calculated value for `RS1` differ, the current value of `RS1` is saved and the constrained value put on the trace.
- **@save, @restore, @update** The value of the `RD` is restored.

ADD The `ADD` instruction executed forward results in *adding* `RS1` and `RS2` and storing the result into `RD`. Execution proceeds even when both source registers hold symbolic values. Not so during backward execution of `ADD`. VIPSTER can not reason about an `ADD` instruction for which both operands are symbolic. There is most likely more than one way to constrain two symbolic intervals and staying complete would require exploring all options. This case is detectable and VIPSTER aborts constraining the current path further, giving detailed information about the cause. Therefore an important step is to first determine wether the reasoning remains sound. When this is the case, the engine effectively falls back to an inverse `ADDI` instruction.

- **@prepare** The solver determines which register is concrete and which is symbolic. Further, it obtains the original operand values for both source register and performs the calculation.
- **@save @constrain @update?:** If original and calculated value for the symbolic register differ, the current value of that register is saved and the constrained value is put on the trace.
- **@save, @restore, @update** The value of the `RD` is restored.

SUB Forward execution of the `SUB` instruction *subtract* `RS2` from `RS1` and stores the result in `RD`. For the inverse semantics of this instruction two things have to be considered. First, during backward execution the engine requires one source register to be concrete in order to remain sound. And second, other than addition, subtraction is not a commutative operation. As a result, the inverse semantics of the `SUB` instruction differs, depending on which source register is symbolic.

- **@prepare** The solver obtains the original operand values for both source registers. It determines which register is concrete and which is symbolic and performs the according calculation (Table 1).
- **@save @constrain @update?:** If original and calculated value for the symbolic source register differ, the current value of the symbolic source register is saved and the constrained value put on the trace.
- **@save, @restore, @update** The value of the `RD` is restored.

Forward Semantics	**Constrained Result**	**Inverse Semantics**
$[a,b]+[c,c]=[a+c,b+c]$	$[a'+c,b'+c]$	$[a'+c,b'+c]-[c,c]=[a',b']$
$[c,c]+[a,b]=[c+a,c+b]$	$[c+a',c+b']$	$[c+a',c+b']-[c,c]=[a',b']$
$[a,b]-[c,c]=[a-c,b-c]$	$[a'-c,b'-c]$	$[a'-c,b'-c]+[c,c]=[a',b']$
$[c,c]-[a,b]=[c-a,c-b]$	$[c-a',c-b']$	$[c,c]-[c-a',c-b']=[a',b']$

Table 1: This table presents the inverse semantics of addition and subtraction as used in the implementation `ADDI`, `ADD` and `SUB`. The inverse semantics shows how to obtain the (constrained) operand. Note that these operations are defined in the work of by Manuel Widmoser.

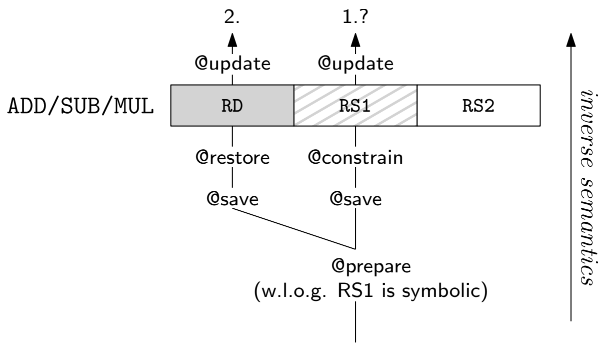


Figure 10: `ADD`,`SUB`, and `MUL` executed and constrained backwards.

MUL The `MUL` instruction during forward execution *multiplies* `RS1` and `RS2` and stores the result in `RD`. At least one source register has to be concrete. If multiplication is performed on a symbolic value the following can be observed:

- the interval representing the result may contain to many values
- the values that can result from multiplication are multiples of the constant factor

Interpreted during backward execution `RD` is *divided* by the concrete source register. The implications of the misleading representation is explained using an example.

Example Assume intervals $[a, b]$ with $0 \leq a \leq b \leq 256$. Following `MUL` instruction was interpreted during forward execution: $[2, 8] * [2, 2] = [4, 16]$. In the current backward pass the result has been constrained to $[11, 15]$. Both bounds are not a multiple of the constant factor.

Dividing $[11, 15]$ by $[2, 2]$ would results in $[5, 7]$ as the constrained value for the source register. Unfortunately, execution has just become imprecise.

Consider forward execution using the confined interval $[5, 7]$ instead of the original interval. $[5, 7] * [2, 2]$ results in $[10, 14]$, but $10 \notin [11, 15]$.

The problem presented in this example is that the lower bound for the constrained source register has been chosen to small and violates the above mentioned property. The lower bound of the constrained result is not a multiple of the concrete factor. The solution is to narrow the interval by rounding the lower and upper bound to a multiple of the constant factor. The lower bound is rounded up, whereas the upper bound is rounded down. Figure 11 illustrates this situation. For the above example the register is then correctly confined to $[6, 7]$ instead of $[5, 7]$.

- **@prepare** The solver obtains the original operand values for both source registers. It determines which register is concrete and which is symbolic.
- **@save @constrain @update:** The symbolic source register is saved and and the possibly constrained interval is calculated(Table 2). The calculated value is put on the trace and the source register is updated.
- **@save, @restore, @update** The value of the `RD` is restored.

Forward Semantics	**Constrained Result**	**Inverse Semantics**
$[a, b] * [c, c] = [a * c, b * c]$	$[(a' * c) - r_1, (b' * c) + r_2]$	$[(a' * c) - r_1, (b' * c) + r_2]/[c, c] =$
		if $0 < r_1$: $[(a' - 1), b'] + [1, 0]$
		if $0 == r_1$: $[a', b']$

Table 2: This table presents the inverse semantics of multiplication as used in the implementation `MUL`. The inverse semantics shows how to obtain the (constrained) operand. In case the lower bound is not a multiple of the constant factor 1 is added to obtain the correct lower bound. Note that this avoids rounding the bounds prior to execution.

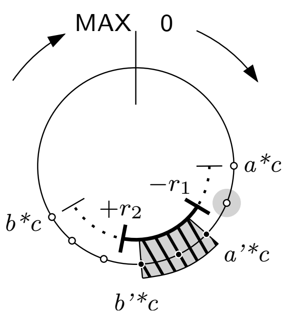


Figure 11: The interval resulting form multiplication $[a*c, b*c]$ is represented by the dashed line. The values that can actually result from multiplication are illustrated as points on the circle. The constrained result found during backward execution is shown as solid line. If the lower bound is not a multiple of the constant factor, division results in the bound $(a' - 1)$ circled in gray. The correct bounds are obtained if the bounds are rounded up/down as shown by the gray-black area.

DIVU The `DIVU` instruction is only defined if the divisor in `RS2` is constant. Executing `DIVU` forward *divides* the dividend in `RS1` by the divisor in `RS2` and stores the quotient in `RD`. Additionally the remainder is remembered on the trace. Interpreting `DIVU` backwards the engine *multiplies* the quotient `RD` and divisor `RS2` to obtain the dividend `RS1`. To ensure that the above property holds, the engine has to also consider remainders when computing the bounds. If a bound in `RD` has not been constrained the initial bound of `RS1` is restore by adding the remainder. In case the upper bound has been constrained it has to be adjusted to account for a possible remainder. The adjustment is based on the following observations:

- Euclidean division is used to calculate upper and lower bounds.
- Euclidean division with divisor c maps c different values to the same quotient.
- Multiplying a quotient and a divisor results in the smallest value (dividend) that maps to that quotient when divided by that devisor.

The computed upper bound for the dividend is the smallest value that maps to the constrained bound of the quotient. There are $c - 1$ values greater than the computed bound that map to the same constrained bound of the quotient. Hence the upper bound is widened by $c - 1$. The lower bound is already correct. It is the smallest value that maps to the dividend.

- **@prepare** The original operand values (dividend in `RS1` and divisor in `RS2`) and the remainder stored during forward execution are obtained.
- **@save @constrain @update:** `RS1` is saved and a new value for it is calculated. The possibly constrained `RD` is multiplied with `RS2` and the bounds are corrected(Table 3). The calculated value is put on the trace and `RS1` is updated.
- **@save, @restore, @update** For `DIVU` the value or `RD` is not restored as selfie only compiles with `RS1` == `RD` and the value for `RS1` is already updated.

Forward Semantics	**Constrained Result**	**Inverse Semantics**
$[a, b]/[c, c] =$ $[(a - r_1)/c, (b - r_2)/c]$	$[(a' - l_1)/c, (b' - l_2)/c]$	$[(a' - l_1)/c, (b' - l_2)/c] * [c, c] =$ $[a' - l_1, b' - l_2] + [l_1, l_2] ==$
$r_1, r_2 < c$	$l_1, l_2 < c$	$\begin{cases} l_1 == r_1, & \text{for } a' == a \\ l_2 == r_2, & \text{for } b' == b \\ l_1 == 0, & \text{for } a' \neq a \\ l_2 == (c-1), & \text{for } b' \neq b \end{cases}$

Table 3: This table presents the inverse semantics of division as used in the implementation DVIU. During forward execution the remainders is 'forgotten'. The constrained result does not account for them either. The inverse semantics has to consider all possible remainders when calculating the bounds. If the lower bound has been constrained the lowest value (remainder 0) is set as the bound. In case the upper bound has been constrained the highest value (remainder $c - 1$) is used. In case a bound has not been constrained the original remainder is restored.

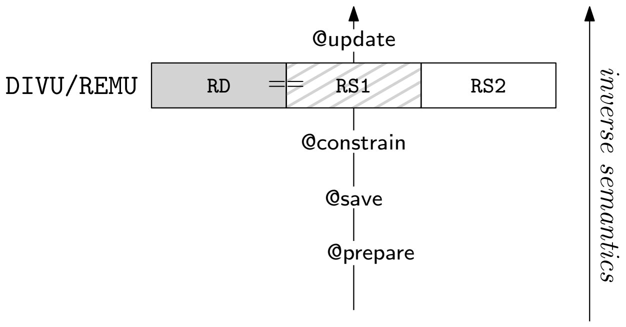


Figure 12: DIVU and REMU executed and constrained backwards.

REMU The REMU instruction interpreted by the forward execution engine performs the *modulo* operation with concrete devisors. Additionally the quotient is remembered on the trace. This instruction is, however, only defined on a subset of symbolic intervals that satisfy certain properties. VIPSTER can only interpret REMU if the symbolic interval is not wrapped *and*

- the cardinality of the interval is strictly greater than the devisor *or*
- the cardinality is less or equal the divisor and the interval contains only values that have the same quotient when divided by the divisor

In the latter case no two values in the interval are congruent modulo c and the result of the modulo operation is an interval. The inverse semantics of the REMU instruction is defined for a even smaller subset. It is defined only if the second case was observed during forward execution. Since RD describes a set of residue classes rather than a set of values, constraining RD means removing a residue class from the set. Propagating the constraint to the source register would require to remove every value belonging to the cut residue class. In general this can not be done as it would split the interval. The only case for which remu is defined is shown in Table 4.

- **@prepare** The original operand values (dividend in `RS1` and divisor in `RS2`) and the quotient stored in course of forward execution are obtained and it is checked wether `REMU` can be executed backwards (is defined).
- **@save @constrain @update:** `RS1` is saved before the new interval is computed (Table 4) and stored on the trace. Afterwards `RS1` is updated.
- **@save, @restore, @update** Selfie compiles `REMU` with `RD == RS1` and hence `RD` is already updated.

Forward Semantics	**Constrained Result**	**Inverse Semantics**
$[a, b] \mod [c, c] =$	$[a' \mod c, b' \mod c]$	
$[a \mod c, b \mod c]$		
$a/c == b/c == q,$		
$r, i \geq 0, r + i < c:$		
$a = qc + r$	$a' = qc + (r + \alpha)$	
$b = qc + (r + i)$	$b' = qc + ((r + i) - \beta)$	
$[qc + r, qc + (r + i)] \mod [c, c] =$		$[r + \alpha, (r + i) - \beta] + [qc, qc]$
$[r, r + i]$	$[r + \alpha, (r + i) - \beta]$	$= [a', b']$

Table 4: $[a, b]$ is an interval that is not wrapped and c the modulo. Additionally, the cardinality of $[a, b]$ is less or equal c and $a/c == b/c == q$ (q is the quotient). Note that no two values in $[a, b]$ are congruent modulo c. The interval $[a, b]$ describes the set $\{a, a + 1, ..., b\}$. and this set can be written as $\{qc + r, qc + (r + 1), ..., qc + (r + i)\}$ with $r, i \geq 0, r + i < c$. The result of the modulo operation is the interval $[r, r + i]$ describing the set $\{r, r + 1, ..., r + i\}$. Now consider the constrained interval $[r + \alpha, (r + n) - \beta]$. In this special case the original interval is not split. Figure 13 illustrates the mapping between these two sets and shows how cutting a residue class removes operand values on the edge of the interval. The constraint is propagated to the source register by calculating $a' = qc + (r + \alpha)$ and $b' = qc + ((r + i) - \beta))$.

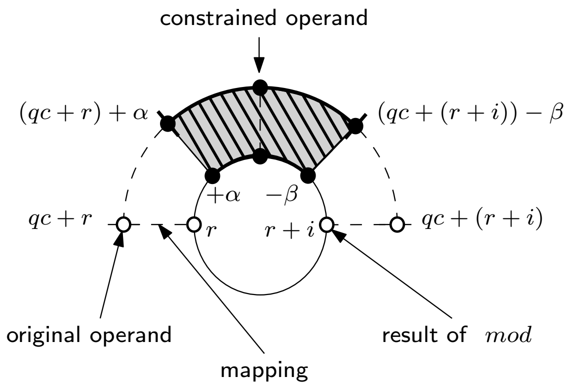


Figure 13: This figure illustrates the bijective mapping between the result of the modulo operation and its original symbolic operand. This is the only case VIPSTER can reason about during backward execution.

SLTU As mentioned the forward semantics of `SLTU` instruction is complex and described in great detail in the work of Manuel Widmoser. `SLTU` interpreted by the forward engine compares `RS1` and `RS2` to determine wether `RS1` < `RS2` holds. One source register is required to be concrete. Before executing `SLTU` the engine will put every feasible branch on the trace. and modify the state to ensure that one of two cases is applicable:

1. `RS1` < `RS2` before execution and the concrete value 1 is stored in `RD` after execution
2. `RS1` $\geq$ `RS2` before execution and the concrete value 0 is stored in `RD` after execution

The inverse semantics of the `SLTU` instruction is simple because no constraint has to be propagated. All that has to be done is restore `RD` and skip the remaining constraints.

- **@prepare** Nothing to do aside from putting the program counter on the trace.
- **@save, @restore, @update** The value of the `RD` is restored and additional trace entries are skipped.

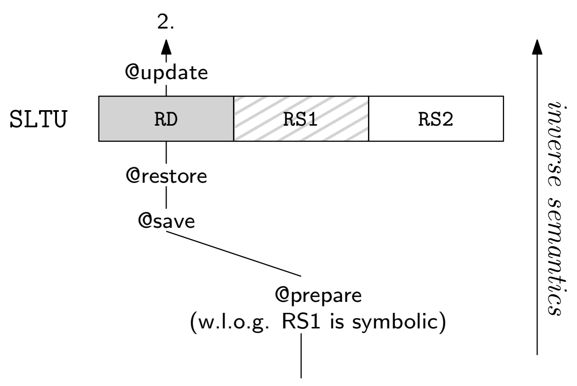


Figure 14: `SLTU` executed and constrained backwards.

3.2.4 Inverse Semantics of Control Flow Instructions and LUI

Control flow instructions and `LUI` do not operate on symbolic intervals and therefore their forward and inverse semantics is straight-forward.

JAL and JALR `JAL` and `JALR` during forward execution modify the program counter and store a return address in `RD`. The program counter and addresses used are always concrete. In course of backward execution the program counter is automatically restored to its previous value. What remains to do is restoring `RD`.

- **@prepare** Nothing to do aside from putting the program counter on the trace.
- **@save, @restore, @update** The value of the `RD` is restored.

BEQ The selfie compiler only emits `BEQ` instructions that use `REG_ZR` as one of the operand registers. `BEQ` instructions are only used in combination with a preceding `SLTU` instruction. This is ensured by requiring each logical expression to contain at least one comparison operator. The second operand register holds the result of the evalution of the `SLTU` instruction. Therefore both operand registers are concrete.

Forward execution of `BEQ` only modifies the program counter. The trace entry for `BEQ` is empty aside from recording the program counter. Since the program counter is read and restored this instruction is ignored during backward execution.

LUI The `LUI` instructions executed by the forward execution engine *loads* an immediate value into `RD`. It does not operate on symbolic values and hence its backward semantics is simple. During backward execution the value or `RD` is restored.

- **@prepare** Nothing to do aside from putting the program counter on the trace.
- **@save, @restore, @update** The value of the `RD` is restored.

3.2.5 Inverse Semantics of Memory Instructions

Memory instructions *transfer* values between memory and registers. All instructions analyzed so far, whether they are executed forward or backward, modify registers. As a result, constraining only happens on register level. This is in principle the issue behind *Register Memory Gap*. The inverse semantics of `LD` and `SD` as defined below bridges this gap.

LD During forward execution the `LD` instructions *loads* the value stored at the specified address in memory into the destination register. The address to load from has to be a constant. The inverse semantics of `LD` is based on three properties of the `LD` instruction:

1. *before* executing `LD` forward the interval in memory represents all concrete values that could be loaded into the destination register
2. *executing* `LD` does not change the value in memory
3. *after* executing `LD` forward the value in the destination register is equivalent to the value stored in memory

It follows from the third property that if in the course of backward execution the destination register has been constrained, the memory value should be equivalent to that constraint value. However, it also follows from the first two properties, that *before* executing `LD` backwards, the interval in memory represents all possible values that can be loaded. *Intersection* of intervals, which is defined in Section 3.2.6, is used to constrain the value in memory.

- **@prepare** The solver obtains the address to load from from the source register. Validity checks are redundant and are therefore omitted.
- **@save @constrain @update?:** If the value currently in memory differs from the register value to be stored, the memory value is constrained using intersection. The value in memory is saved and the memory value is updated.
- **@save, @restore, @update** The value of the `RD` is restored.

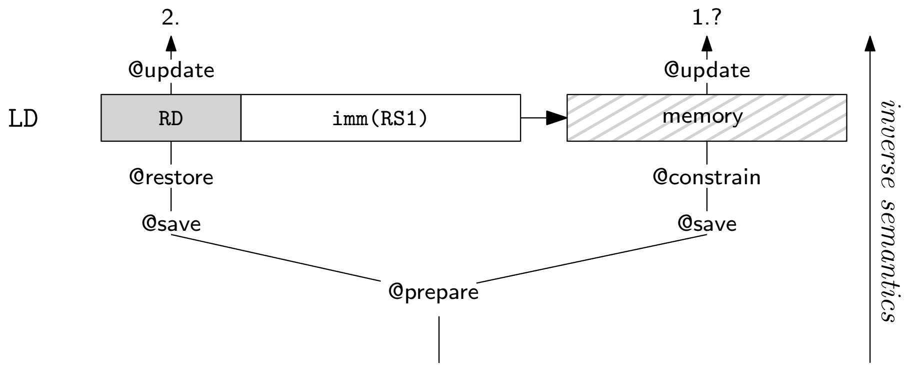


Figure 15: `LD` executed and constrained backwards.

SD Executed forward the `SD` instruction *stores* a register value at a specified address in memory. The inverse semantics of `SD` is based on similar properties as the `LD` instruction above:

1. *before* executing `SD` forward the register value represents all concrete values that could be stored in memory
2. *executing* `SD` does not change the value of the register
3. *after* executing `SD` forward the value in the register is equivalent to the value now in memory

If backward execution has constrained the value in memory, it follows from the third property, that the register value should be equivalent to that value. But as a consequence of the first two properties, *before* executing `SD` backwards, the register value represents all possible values that can be stored. Again *intersection* is used to constrain the register value.

- **@prepare** The solver obtains the address to store at from the source register. Validity checks are redundant and are therefore omitted.
- **@save @constrain @update?:** If the current register value differes from value in memory, the register value is constrained using intersection. The register value is saved and updated.
- **@save, @restore, @update** The value in memory is restored.

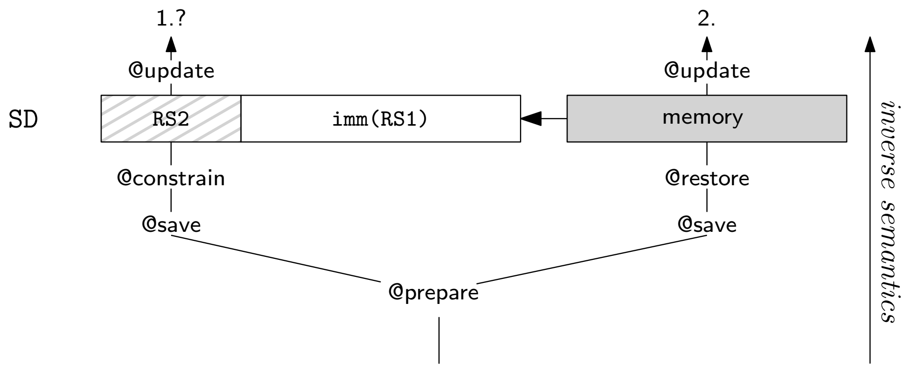


Figure 16: `SD` executed and constrained backwards.

3.2.6 Interval Intersection

Memory instructions constrain symbolic values by calculating the intersection of two intervals [9], [10]. Union and intersection of circular intervals is not closed under those operations in general. However, if certain properties hold, the result of an intersect is again an interval. These properties allow the definition of four cases VIPSTER can reason about.

Definition 5 (Intersection)**.** *The intersection of two intervals* $[a, b]$ *and* $[c, d]$*, with* $a < b$ *and* $c < d$*, is defined as*

$$[a, b] \cap [c, d] = [\max(a, c),\ \min(b, d)]$$

The intersection is empty if

$$(a > d) \lor (b < c)$$

Corollary 2. *The intersection of two intervals* $[a, b] \cap [c, d]$ *is empty if and only if* $\max(a, c) > \min(b, d)$*.*

Case 1: No wrap-around In the simplest case, the definitions and corollary can be used without any further reasoning to calculate the intersection.

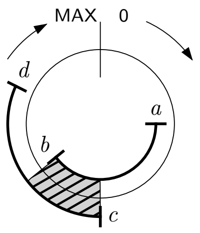


Figure 17: Interval intersection with two intervals that have not wrapped around.

Case 2: Single wrap-around These cases describe the intersection of a wrapped interval and interval that is not wrapped. Figure 18 shows the four different scenarios in which this case could occur. VIPSTER can only reason about three out of those four cases correctly. We call the intersections VIPSTER can reason about correctly *valid* and the intersection VIPSTER cannot reason about *invalid*. An intersection is *valid* if the result is again a closed or empty interval as illustrated by case (a), (b) and (c) of Figure 18. Otherwise the intersection is *invalid* as illustrated by case (d). An invalid intersection is not empty and the result cannot be written as closed interval.

Let $a > b \land c \leq d$, then

$$[a, b] \cap [c, d] = \begin{cases} [a, \min(b, d)], & \text{for } d < a \land c < b \ \ldots \ \text{(a)} \\ [\max(a, c), d], & \text{for } c > b \land d > a \ \ldots \ \text{(b)} \\ [\max(a, c), \min(b, d)], & \text{otherwise} \ \ldots\ldots\ldots \ \text{(c), (d)} \end{cases}$$

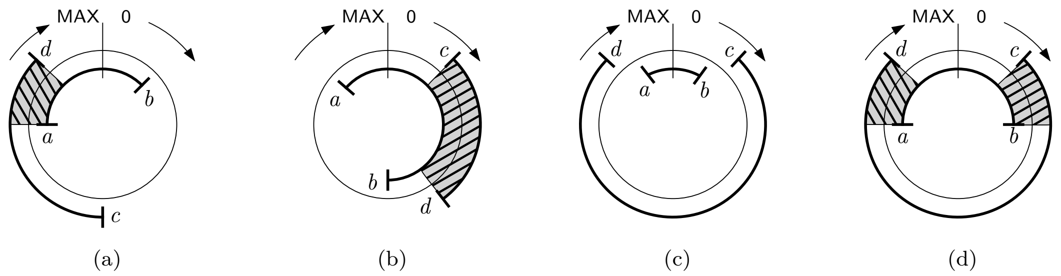


Figure 18: The wrapped interval $[a, b]$ can be written as $[a, MAX - 1] \cup [0, b]$. An intersection is valid if the not wrapped interval $[c, d]$ intersects exclusively either with $[a, MAX - 1]$ as shown by (a) or $[0, b]$ as pictured by (b). The intersection is also valid if the two intervals do not intersect as in case (c). The only invalid case is (d) where $[c, d]$ intersects with both parts of the wrapped interval.

Case 3: Double wrap-around Intersecting two wrapped intervals results in a wrapped interval.

Let $a \geq b \wedge c \geq d$, then

$$[a, b] \cap [c, d] = [\max(a, c), \min(b, d)]$$

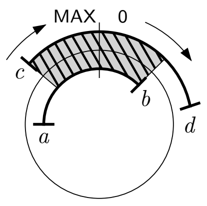


Figure 19: The intersection of two wrapped intervals

Case 4: Full interval This special case uses the concept of *cardinality of intervals* as defined in the work of Manuel Widmoser. Informally defined, a full interval contains every value from 0 to MAX, independent of its representation. The result of intersecting an interval with a full interval is always that other interval.

Let $a < b$, then

$$[a, b] \cap [0, MAX] = [a, b]$$

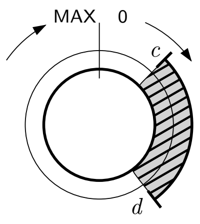


Figure 20: Intersection with a full interval.

3.2.7 Inverse Semantics of System Calls

System calls provide a means by which the program *requests a service.* Selfie features five system calls, `open`, `read`, `write`, `malloc` and `exit`. When the program is executed symbolically those system calls are handled by VIPSTER. Performing the requested service has side effects regarding the environment in which the program is executed. The inverse semantics of a system call does not consider these side effects, whith exception of the `read` syscall.

ECALL open, write, malloc, and exit Executed during forward execution the `ECALL` instruction signals VIPSTER that the program executed requests a service. The arguments provided have to be constants. VIPSTER then executes the requested service on behalf of the program and places a response in register `A0`. Depending on the performed service the response is the file descriptor of the opened file or the number of bytes read, written or allocated. As every side effect is ignored during backward execution, it is only the register containing the response that gets restored

- **@prepare** Nothing to do aside from putting the program counter on the trace.
- **@save, @restore, @update** The value of the `A0` is restored.

ECALL read During forward execution the `read` syscall *introduces* symbolic values of arbitrary size and stores them in memory at a specified address. VIPSTER creates intervals of maximal cardinality, stores them on the trace and creates the mapping into memory. The arguments for the `read` call, size, buffer, and file descriptor have to be concrete. The number of symbolic values and increases it upon a `read`. The inverse semantics of the `read` syscall is to *remove* the possibly constrained values from memory and restore the previous values. Additionally the number of symbolic values is decreased.

But before removing the values from memory, the satisfiability of the current path in regard to these values is checked. At the current point of Backward Execution the values in memory reflect all constraints associated with them. If no interval introduced by the `read` call is `UNSATISFIABLE` the constraint solver has found values that satisfy the path condition. An `UNSATISFIABLE` interval means that no value satisfies the path condition and hence that the `ERROR` is unreachable.

The constraining process is finished when no symbolic values remain. If the solver reaches this point without finding and `UNSATISFIABLE` interval introduced by `read` the `ERROR` is reachable and witnesses that satisfy the path condition have been found.

- **@prepare** Checking the satisfiability of each value read into memory ($\neq$`UNSATISFIABLE`). Note that the provided address refers to the first trace entry stored by the forward engine (restore in right order).
- **@save, @restore, @update** For each read value, the current (constrained) value is saved before restoring the value that was present in memory before `read` was executed. Each memory location is updated.
- **@save, @restore, @update** The value of the `A0` is restored.

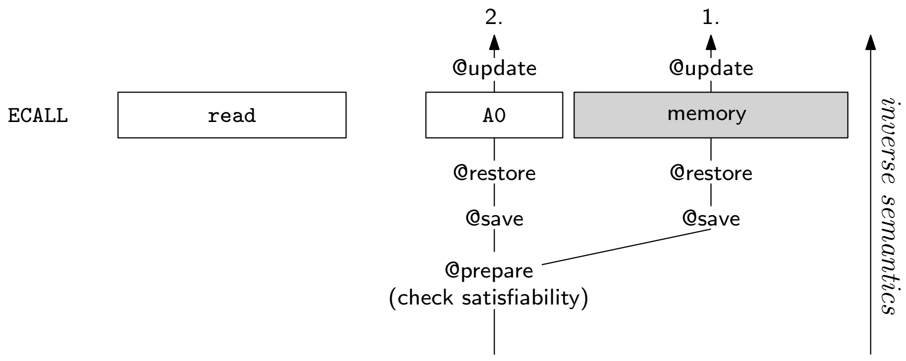


Figure 21: `ECALL read` executed and constrained backwards.

3.3 Backward Execution - Reachability

The previous section presented the constraint solving aspect of backward execution. It described how constraints collected on a trace are propagated backwards and how and when intervals are constrained. The main objective was to find witnesses for the given path if it is satisfiable. But so far no explicit explanation was given as to why a path explored can be unsatisfiable. Only the reason was mentioned, a simple but lazy execution engine. This section will explain the reasons.

3.3.1 Register Memory Gap

This phenomenon is simply a consequence of how computers with load–store architecture work. A value reside in memory until it gets loaded into a register where it is used in arithmetic operations, manipulated or compared by instructions executed by the processor. A manipulated value is then often stored back into memory. Not so a value that was used as an operand in a computation or a comparison instruction.

VIPSTER's machine model is based on `RISC-V` instruction set architecture, a load-store architecture as described above. The execution engine interprets instructions and constrains intervals on register level, but it does not reflect the constraints onto the original value in memory. Hence the name Register Memory Gap.

Listing 1: Register-Memory-Gap

```
 1  // assume: *x is symbolic [0, 255]
 2
 3  uint64_t main() {
 4
 5    if (*x > 7) {
 6      ...
 7      if (*x < 3) {
 8        // not satisfiable
 9        return DEVISION_BY_ZERO;
10      }
11      ...
12    }
13    return 0;
14  }
```

Listing 2: Instructions of Listing 1

```
 1  (~5): LD   $t0,   -8($gp)
 2  (~5): LD   $t0,    0($t0)
 3
 4  (~5): ADDI $t1, $zr,    7
 5  (~5): SLTU $t0, $t1,  $t0
 6  (~5): BEQ  $t0, $zr, addr
 7
 8
 9  (~7): LD   $t0,   -8($gp)
10  (~7): LD   $t0,    0($t0)
11
12  (~7): ADDI $t1, $zr,    3
13  (~7): SLTU $t0, $t0,  $t1
14  (~7): BEQ  $t0, $zr, addr
```

In Line 5 the execution engine would loads the interval associated with `x` into register and compares it with 7 using `SLTU`. In this case `SLTU` causes execution to fork and the interval will be constrained in register. Assuming the true branch is explored, in Line 7 `x` will be loaded again, this time to be compared to 3. Again, the interval present in register is constrained and the true branch, an unsatisfiable branch, can and will be followed.

Addressing this issue during forward execution is of course possible but tedious. Every time a interval, that has likely been modified between loading and comparing, is constrained in register the constraint has to be properly propagated into memory. Properly propagating the constraints is exactly what Backward Execution does, but instead of going back and forth during forward execution it suffices to go back once. The key instructions that propagate the constraints into memory are `LD` and `SD`. Simon Bauer shows the this in his thesis with a detailed example.

3.3.2 Aliasing and Dependent Variables

In computing, aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program.

In mathematics and computing, a dependent variable is a variable whose value *depends* on one or more independent variables. If an independent variable changes, then the dependent variable is affected by that change.

Independent variables are factors that change independent of other variables. They represent arbitrary input, cause or reason, whereas dependent values stand for output or outcome. The most common symbols for input and output are respectively x and y and the function $y = f(x)$ shows the relation between an independent variable x and the dependent variable y.

VIPSTER symbolically executes RISU-U binary code and there are no variables in binary code, only memory locations. Unfortunately, this concept of dependencies is still applicable. Dependent variables translate directly to dependent memory locations as this code shows.

Listing 3: Aliasing

```
 1  // assume: *x is symbolic
 2
 3  uint64_t main() {
 4
 5    uint64_t* y = x;
 6
 7    if (*x > 3) {
 8      ...
 9      if (*y < 3) {
10        // not satisfiable
11        return DEVISION_BY_ZERO;
12      }
13      ...
14    }
15    return 0;
16  }
```

Listing 4: Dependent Variables

```
 1  // assume: *x is symbolic
 2
 3  uint64_t main() {
 4
 5    uint64_t y = *x;
 6
 7    if (*x > 3) {
 8      ...
 9      if (y < 3) {
10        // not satisfiable
11        return DEVISION_BY_ZERO;
12      }
13      ...
14    }
15    return 0;
16  }
```

During forward execution Line 5 in Listing 4 introduces a dependent variable `y` whose value depends on the independent variable `*x`. Note that `*x` and `y` refer to different memory locations. The comparison in Line 7 constrains `*x`. The dependent variable `y` should respond to that change but the execution engine ignores any dependencies among variables. When the engine loads `y` in Line 9 `y` does not reflect the constraint.

In Listing 3 the same example is shown with pointer aliasing. Since `x` and `y` refer to the same memory location, constraining `*x` in memory during backward execution (Register Memory Gap!) also constrains `*y`.

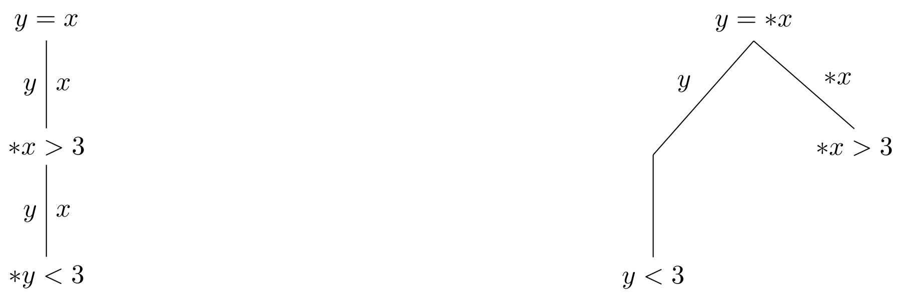


Figure 22: On the left side pointer aliasing is shown. Since x and y refer to the same memory location this is not an issue.
The right illustration shows a variable y that depends on the value of $*x$. The dependency is resolved at the root of this tree.

The issue of dependent could also be addressed during forward execution, but the process this time would be even more tedious. Aside from propagating the constraint back into memory of the directly affected interval, the constraint has to also be propagated to every dependent variable. In the most straight forward way this requires propagating the constraint back to the creation of the dependent variable, constraining it and execute forward to apply the effect of the constraint.

Also in this case Backward Execution solves the problem going back once. The dependency amongst the variables is resolved during Backward Execution at the point it was created during forward execution. It is the `LD` and `SD` that are the key instruction in this process. Figure 22 shows the above example as tree, visualizing the dependency. Simon Bauer provides an detailed example in his thesis explaining the process.

3.4 Path Exploration

VIPSTER's execution engine explores paths using a depth-first approach. So far it was outlined how the engine explores a single path and builds a trace of execution. The constraint solver then reasons about satisfiability and attempts to generate concrete witnesses for this path. In course of this the solver builds a trace of backward execution.

The last missing piece that enables us to explore all feasible paths is VIPSTER's traversal engine. It uses a back-tracking algorithm to revert to a point in execution at which the path diverged, but where one path has not been explored. As mentioned before, the only instruction that can cause a path do diverge is `SLTU` and only so during forward execution. Therefore each `SLTU` instruction on the forward trace represents a possible branching point and the engine will go back to the most resent one.

In order to revert to a previous `SLTU` and still be able to correctly continue execution along a different path the traversal engine operates in two steps.

- The engine first restores the execution state that was observed right before the `SLTU` was executed, but after it has been prepared (constraints). At this point it `RS1` $<$ `RS2` or `RS1` $\geq$ `RS2`. Restoring is achieved by rewinding backward and forward execution. Starting execution at this point would lead the engine to explore the same path again, as the result of `SLTU` would be the same.
- To prompt the engine to follow a different path the traversal engine modifies the state. It switches the constrained intervals. Executing `SLTU` now will generate a different result.

3.4.1 Restore Execution State

A previous execution state is restored by rewinding first backward then forward execution.

Abstract Idea Each trace entry represents a change in state and no modification is missed by the trace. The change is so small that a trace entry also recorded a *before*, a *after* and a *where*. By simply walking the trace backwards and restoring the *befores* every change is reversed. This is how execution can be rewinded to an any previous state.

Based on this idea, the *undo* semantics for every instruction is defined. Aside from the `SLTU` and `ECALL` instruction it makes little difference whether the instruction to be undone was executed forward or backward.

LUI, JAL, JALR and ECALLS These instructions executed forawrd and backward operate on concrete values and only modify the destination register.

- **@restore, @update** The value of the `RD` is restored.

ADDI, ADD, SUB and MUL During forward execution arithmetic instructions modify the destination register and during backward execution they restore the destination register. During backward execution a source Register might have been constrained.

- **@restore, @update** The value of the `RD` is restored.
- **@restore, @update?** The value before constraining is restored.

DIVU and REMU Aside form modifying the destination register these arithmetic instructions also store an additional value (remainder, divisor) on the trace.

- **@restore, @update** The value of the `RD` is restored and additional trace entries are removed from the trace.

LD and SD The `LD` instruction modifies `RD` and potentially constrains a memory location during backward execution. whereas the `SD` instruction modifies a memory location an possibly constrains `RD` .

- **@restore, @update** For `LD` the `RD` is restored and for `SD` the memory location is restored.
- **@restore, @update?** For `LD` the memory constraint is removed and for `SD` the register constraint is removed.

ECALL read During forward execution the `read` ecall pushes symbolic intervals on the trace, stores the trace counters in memory and increases the number of symbolic variables. Register `A0` is modified last and contains the number of bytes read.

Backward execution removes the read values, restores the previous values and then restores register `A0`. Executing read backwards decreases the number of symbolic values.

Undoing the effects of `read` requires to distinguish between forward and backward execution for two reasons. First, to know whether to increase or decrease the number of symbolic variables and second to find the number of bytes read(either `A0` or previous `A0`).

- **@restore, @update** Register `A0` is restored.
- **@restore, @update** The values before/after the read are restored.

SLTU During forward execution all feasible branches are stored on trace and execution along one branch. The constraint solver ignores all other the branches and only restores `RD`.

The traversal engine distinguish between forward and backward execution of `SLTU`:

- An `SLTU` that was executed by the constraint solver only requires restoring `RD`.
- An `SLTU` that was executed by the execution engine indicates a potential branching point. If another branch was pushed on the trace the traversal engine prepares that branch before it turns over control to the execution engine. Figure 23 explains this process.
- **@restore, @update** `RD` is restored and the symbolic source register is assigned the next branch (Figure 23)

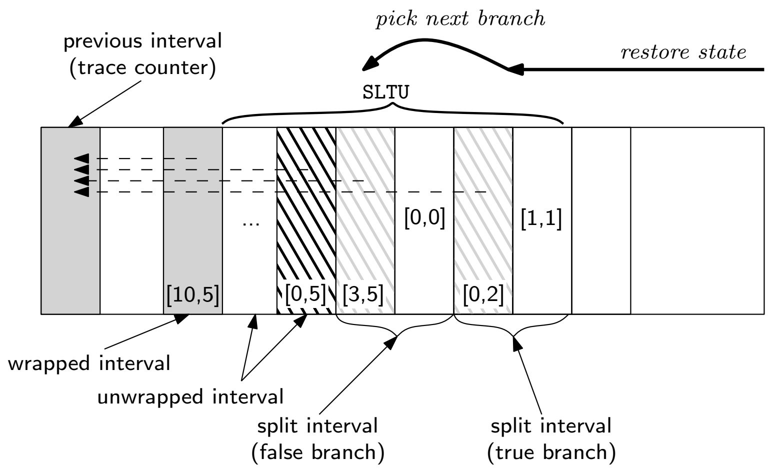


Figure 23: This figure shows the trace entries as they can be created in course of executing an `SLTU`. We use [10,5] < [3,3] in this example and assume `RS1` is the symbolic interval.
If the interval that is compared by the `SLTU` instruction is wrapped, the engine unwraps the interval, creating two intervals that both point to the same 'previous' interval as the original wrapped interval. This illustration only shows one unwrapped part of the wrapped interval([0,5]). In the illustrated case the unwrapped interval is split into two intervals, [3,5] representing the false branch and [0,2] representing the true branch. The intervals [0,0] and [1,1] are the results of evaluating `SLTU`. During forward execution the value in `RS1` is [0,2] and `RD` holds [1,1] and the engine first explores the true branch.
The traversal engine when reaching this `SLTU` instruction checks the trace for additional entries and in this case finds an unexplored false branch. It picks the next branch by storing the corresponding interval in `RS1`. Here the false part of the split interval [3,5] is stored in `RS1`. At this point it returns control to the execution engine. The engine will evaluate `SLTU` and continue execution along the false branch.

4 Conclusion

The execution engine ignores issues that arise during symbolic forward execution, namely the Register Memory Gap and dependencies amongst variables. As a consequence the execution engine may report false positives.

To address these issues, I developed the constraint solver that I presented in this thesis. The solver I developed uses *Backward Execution* to reason about the reachability of errors and to generate concrete witnesses for paths on which the errors occurred. I defined and explained the *symbolic inverse semantics* of instructions that is used for Backward Execution, including the inverse semantics of arithmetic instructions that propagate constraints and memory instructions that constrain symbolic values in memory.

The traversal engine I implemented and presented last in this thesis uses a simple depth-first search algorithm that rewinds backward and forward execution and enables us to explore all feasible paths.

4.1 Future Work

Vipster's reasoning is 64-bit-precise. I already developed a first approach towards byte-precise reasoning that seems promising for simple examples. Basically, the idea is to interpret the 64 bits not as one machine word, but as four independent bytes.

At this stage it is to early too determine whether the idea is viable or not. It could, however, be interesting to pursue this approach further.

Idea of Masking Registers When register values are operated on (ex. `SLTU`, `SD`,...), often only certain bits of the initial memory value determine the outcome of the operation. If register values are shifted by multiplication and/or division before an operation is performed, the result of the operation depends only on the bits that where *not* shifted out.

Based on this observation, my idea was to use a bit mask with byte granularity for each register to determine which bytes of the initially loaded memory value are considered in an operation. In other words, the mask keeps track of the bytes that have not been shifted in our out. Upon a `LD` and `SD` only these bytes are considered when constraining symbolic values.

References

[1] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1066–1071, May 2011.

[3] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[4] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, pp. 82–90, Feb. 2013.

[5] K. Sen, “Concolic testing,” in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, (New York, NY, USA), pp. 571–572, ACM, 2007.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, May 2012.

[7] C. 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, (New York, NY, USA), pp. 198–213, ACM, 2017.

[8] A. Niemetz, E. Univ, B. Armin, A. Zweitbeurteiler, Prof, C. W. Barrett, J. Kepler, T. Wissenschaften, and I. Zusammenfassung, “Bit-precise reasoning beyond bit-blasting,”

[9] A. Gotlieb, M. Leconte, and B. Marre, “Constraint solving on modular integers,” in *ModRef Worksop, associated to CP’2010*, (Saint-Andrews, United Kingdom), Sept. 2010.

[10] T. Hickey, Q. Ju, and M. H. Van Emden, “Interval arithmetic: From principles to implementation,” *J. ACM*, vol. 48, pp. 1038–1068, Sept. 2001.