Witness Extraction and Validation in Unicorn

Haslauer Bernhard
Advisor: Christop Kirsch

Paris Lodron University Salzburg, A-5020 Salzburg, Austria
Department of Computer Science

July 20, 2024

Abstract. Unicorn, a toolchain for symbolic execution, was previously limited to evaluating only whether a given machine code could run into an error state without providing the corresponding inputs for such a case. To determine these inputs, a series of steps with external tools have to be run through. This Bachelor's thesis explores the possibility of incorporating this feature directly into Unicorn, with the objective of improving Unicorn's capabilities in generating and validating the inputs for these errors states. With this proposed extension, Unicorn can extract the inputs and feed them into its built-in RISC-V emulator to validate if these inputs indeed induce the expected errors. If the inputs are accurate, the emulator encounters the predicted errors. In addition, this thesis provides an insight into the concepts used as well as details about the implementation.

Keywords: Unicorn · SAT · SMT · Rust · Symbolic Execution · Program Verification.

Table of Contents

1 Introduction

Symbolic execution (SE) is gaining popularity as a technique in program testing. Over the past few decades, the design and performance of available implementations have seen notable improvements [1]. Current applications are known for their high effectiveness, even though at the cost of computational resources. Unicorn, a project written in Rust and developed at the Paris Lodron University in Salzburg, provides a bit-precise SE tool for a RISC-V machine code. It employs bounded model checking with classical solvers. Although Unicorn does incorporate quantum computing in its broader scope, this thesis will focus exclusively on the classical computing aspects. For readers interested in the quantum computing component, a full discussion about Unicorn provides this paper [9].

Unicorn use a common approach by combining SE, *satisfiability modulo theories* (SMT) solvers as well as *boolean satisfiability problem* (SAT) solvers to find possible errors a program can run into, by translating RISC-V machine code into *finite state machine* (FSM) over bitvectors and arrays and then transform it into semantic-preserving SMT formulae or further into SAT formulae.
The FSM is constructed in such a way that a state S is reachable from the initial state by a finite number n of state transitions if and only if there are inputs to the code that makes a machine reach the state modeled by S after executing no more than n instructions [8].
The satisfiability of the SMT and SAT formulae can than be checked by SMT solvers or SAT solvers. Currently, Unicorn is able to generate BTOR2 and DIMACS files, and connects them directly to Boolector [12] and Z3 [11] along with Kissat [4] and CaDiCaL [4].

For a given machine code, Unicorn returns for every error state if there are any inputs that would lead to that state. However to get the concrete inputs Unicorn is used to generate a *conjunctive normal form* (CNF) file in DIMACS format. Then a SAT solver, like Minisat [14], is used to generate the assignment (SAT file), lastly the information form the CNF and the SAT file are combined to extract the inputs. In addition, the inputs have to be validated by hand.

The first sections of my thesis will cover the fundamental concept of SAT and SMT, as well as the BTOR2 format. Followed that, an exploration of the program language Rust is provided. The core of the thesis lies in the implementation itself. Backed with some practical examples to illustrate the usage. Additional information, including Unicorn Code, will be provided in the appendices.

2 SAT and SMT

The *boolean satisfiability problem* (SAT), is a fundamental problem in computer science and deciding whether a Boolean formula can be satisfied or not It can be defined formally as follows:

Definition 1 (SAT-problem).
For numbers $n, m \in \mathbb{N}$, let there be m clauses with n variables each. A clause is a disjunction of literals x_i or $\neg x_j$ with $i, j \in \{1, \ldots, n\}$. It must be decided whether there exists an assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ of the variables such that all clauses are satisfied, that is, yield the truth value 1.

Theorem 1 (Cook's theorem).
The SAT-Problem is in NP and it is NP-complete.

The Cook's theorem 1 states that the SAT-problem is a problem that can be solved in polynomial time by a nondeterministic Turing machine and every other problem in NP can be transformed (or reduced) into the SAT-problem in polynomial time [16].

While SAT focuses on Boolean problems *satisfiability modulo theories* (SMT) generalizes SAT to more complex formulas over one or more theories. In Unicorn a SMT solver over the theory of arrays is used. This theory was introduced by John McCarthy in [13] and is very useful for soft- and hardware verification, as it can easily model the behavior of, e.g., arrays or memory. The theory consist of two functions *read(a, i)* and *write*. *Read* is used to get an element stored in array a at index i, while write stores the value v in the array a at index i. Formally this model can be described with the following axioms 2, 3:

Definition 2 (Read-write axioms [2]).
$\forall a$: *Array*, $\forall i, j$: *Index*, $\forall v$: *Value:*
$i = j \Rightarrow read(write(a, i, v), j) = v$
$i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j)$

Definition 3 (Extensionality axiom [2]).
for a, b: Array:
$(\forall i$: *Index*: $read(a,i) = read(b,i)) \Rightarrow a = b$

Unicorn, to be exact, makes use of the module over fixed-size bit arrays for the SMT solver. What is a specification of the more general theory of arrays.

The *conjunctive normal form* (CNF) plays a crucial role in SAT and SMT problems. It is a standardized representation of a logical formula, composed of a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. In CNF, literals are either variables or the negation of variables. Modern SAT and SMT solvers are designed to handle problems expressed in CNF, so non-CNF problems must be converted to CNF upfront. But it is proven that it is possible to convert any boolean formula to CNF in linear time. [3]

2.1 Kissat

One of the SAT-solvers supported by Unicorn is *Kissat* [4], a price winning [5] sat solver considered to be an improved reimplementation of CaDiCal [4] in C. It has improved data structures, better scheduling, optimized algorithms, memory use, and more. The development of Kissat represents a significant advance in the state of the art in SAT solving, and it has been used to solve some of the most challenging SAT instances. Therefore Kissat was chosen over CaDiCal and Varisat[6] for the implementation presented in this thesis.
For Unicorn a Rust binding is used to get access to Kissat through safe Rust functions on a Solver type that wraps the actual Kissat solver.

3 BTOR2 Format

The BTOR2 format is a model checking format for capturing hardware in a bit-precise manner. It is line base and simple to parse format and is supported by various model checkers like BtorMC or Boolector. BTOR2 is a generalization of BTOR, which is a format for quantifier-free formulas over bit-vectors and arrays and can model registers and memories [12].
The structure of BTOR2 is a sequence of instructions without forward references or control flow options. An instruction has the structure shown in fig. 1.

identifier	operator	result size	{arguments}

Fig. 1: BTOR2 instruction format.

Intermediate variables, referenced by the unique *identifier*, are used to store the result of an operation. The *result size* specifies the number of bits the result is represented by. The number of *arguments* depends on the *operator*.
The BTOR2 grammar is shown in fig. 2.

```
⟨num⟩      ::= positive unsigned integer (greater than zero)
⟨uint⟩     ::= unsigned integer (including zero)
⟨string⟩   ::= sequence of whitespace and printable characters without '\n'
⟨symbol⟩   ::= sequence of printable characters without '\n'
⟨comment⟩  ::= ';' ⟨string⟩
⟨nid⟩      ::= ⟨num⟩
⟨sid⟩      ::= ⟨num⟩
⟨const⟩    ::= 'const' ⟨sid⟩ [0-1]+
⟨constd⟩   ::= 'constd' ⟨sid⟩ ['-']⟨uint⟩
⟨consth⟩   ::= 'consth' ⟨sid⟩ [0-9a-fA-F]+
⟨input⟩    ::= ( 'input' | 'one' | 'ones' | 'zero' ) ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
⟨state⟩    ::= 'state' ⟨sid⟩
⟨bitvec⟩   ::= 'bitvec' ⟨num⟩
⟨array⟩    ::= 'array' ⟨sid⟩ ⟨sid⟩
⟨node⟩     ::= ⟨sid⟩ 'sort' ( ⟨array⟩ | ⟨bitvec⟩ )
             | ⟨nid⟩ ( ⟨input⟩ | ⟨state⟩ )
             | ⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
             | ⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
             | ⟨nid⟩ ( 'init' | 'next' ) ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
             | ⟨nid⟩ ( 'bad' | 'constraint' | 'fair' | 'output' ) ⟨nid⟩
             | ⟨nid⟩ 'justice' ⟨num⟩ ( ⟨nid⟩ )+
⟨line⟩     ::= ⟨comment⟩ | ⟨node⟩ [ ⟨symbol⟩ ] [ ⟨comment⟩ ]
⟨btor⟩     ::= ( ⟨line⟩'\n' )+
```

Fig. 2: Syntax of the BTOR2 in EBNF. [12]

Btor2 generalizes Btor and extends it by the usage of sorts. The sort keyword is used to define arbitrary bit-vector and array sorts. Furthermore, this allows to specify multi-dimensional arrays and can be extended to support functions, floating points, and more. The format distinguish between node identifiers (nid) and sort identifiers (sid), but doesn't allow any identifier to be in both sets. To declare bit-vector and array variables of a given sort the keyword input is used. Memory and registers can be specified by using the *state* keyword, with the *init* keyword an explicit definition is enabled.

A transaction function for memory and registers can be defined with the keyword *next* and the current and next states as argument. BTOR2 supports also *bad* state properties, invariant *constraints*, as well as the keywords *fair* and *justice* to specify fairness constrains and liveness properties.

Table 1 lists all supported bit-vector and array operators with there respective sorts [12].

indexed		
[su] ext ω	(un)signed extension	$\beta^n \rightarrow \beta^{n+\omega}$
slice u l	extraction, $n > u \geq l$	$\beta^n \rightarrow \beta^{u-l+1}$
unary		
not	bit-wise	$\beta^n \rightarrow \beta^n$
inc, dec, neg	arithmetic	$\beta^n \rightarrow \beta^n$
redand, redor, redxor	reduction	$\beta^n \rightarrow \beta^1$
binary		
iff, implies	Boolean	$\beta^1 \times \beta^1 \rightarrow \beta^1$
eq, neq	(dis)equality	$\mathcal{S} \times \mathcal{S} \rightarrow \beta^1$
[su]gt, [su]gte, [su]lt, [su]lte	(un)signed inequality	$\beta^n \times \beta^n \rightarrow \beta^1$
and, nand, nor, or, xnor, xor	bit-wise	$\beta^n \times \beta^n \rightarrow \beta^n$
rol, ror, sll, sra, srl	rotate, shift	$\beta^n \times \beta^n \rightarrow \beta^n$
add, mul, [su]div, smod, [su]rem, sub	arithmetic	$\beta^n \times \beta^n \rightarrow \beta^n$
[su]addo, [su]divo, [su]mulo, [su]subo	overflow	$\beta^n \times \beta^n \rightarrow \beta^1$
concat	concatenation	$\beta^n \times \beta^m \rightarrow \beta^{n+m}$
read	array read	$\mathcal{A}^{\mathcal{I}\rightarrow\epsilon} \times \mathcal{I} \rightarrow \epsilon$
ternary		
ite	conditional	$\beta^1 \times \beta^n \times \beta^n \rightarrow \beta^n$
write	array write	$\mathcal{A}^{\mathcal{I}\rightarrow\epsilon} \times \mathcal{I} \times \epsilon \rightarrow \mathcal{A}^{\mathcal{I}\rightarrow\epsilon}$

Table 1: BTOR2 operators. β^n represents bit-vectors with size n and $\mathcal{A}^{\mathcal{I}\rightarrow\epsilon}$ are arrays with index sort $\mathcal{I}$ and element sort ϵ [12].

4 Program Language Rust

The program language Rust is developed and maintained by Mozilla since 2009. The language priorities are performance, type safety, and concurrency while enforce memory safety and prevent data races. Rust aims for low level performance while keeping high level safety. As Unicorn is written in Rust a big part was to get familiar with this language. In this section a overview of rust and its features and peculiarities will be provided.

The subsequent section primarily draws upon the official Rust documentation [15] and a book authored by the Rust core developers [10].

4.1 Syntax and Concepts

The syntax of Rust is very similar to C++, although it is highly influenced by the ideas of functional programming. Code blocks are defined by curly brackets and control flow is provided by keywords like *if* , *else* , *while* , and *for* .

(A full list of the Rust keywords is given in the appendix C.)

Match is used for pattern matching. To declare a variable the keyword *let* is used, note that in rust all variables are immutable by default for mutable variables *mut* is needed.
Functions are declared with the keyword *fn* and can be anywhere in the scope where the caller can see it. The type of the parameters must be declared. Return types are declared after ->.
In Rust, the return value is equal to the final expression in the function body, although with the *return* keyword a function can return early. Therefor there are no semicolons at the last expression in the function body, adding a semicolon to the end turns the expression to a statement and the function will not return the value.

```
fn example_function(parameter1: i32, parameter2: char) -> bool {
  println!("the value of {parameter2} is {parameter1}");
  true
}
```

Rust, other than most languages, distinct between statements and expressions. *Statements* are instructions that perform some action and do not return a value where *expressions* evaluate to a resultant value.

Therefor you can not assign a statement to a variable and this code snippet will run into an error.

```
let x = (let y = 6);
```

This is different to other languages like c or ruby. There the line $x = y = 6$ assigns to both variables x and y the value 6.

Expressions evaluate to a value and make up the most code in Rust. Math operations like $3+1$ are expressions. Expressions can be part of a statement, the 6 in `rust` let y = 6 is an expression. Calling a function is an expression. A new scope block is also an expression.

```
let f = {
    let x = 3;
    x + 1
};
```

4.2 Data Types

In Rust Data types are statically, that means the Rust compiler must know the data types of all variables at compile time.
The rust compiler is usually able to estimate the desired variable type based on the value, usage, and context. In case the type is not decidable, the compiler will display an error. This section will cover two data type subsets: *scalar* 4.2.a and *compound* 4.2.b.

4.2.a Scalar Type represents a single value. Similar to other languages rust uses the four primary types: integers, floating-points, Booleans, and characters. Rust supports unsigned and signed integers with a explicit size, e.g. `u32` is a 32-bit unsigned integer (signed integers start with *i* instead of *u*).
Floating-points are either `f32` or `f64`, both are signed.
Booleans are one byte in size and chars are four bytes and represent a Unicode scalar Value instead of just ASCII.

4.2.b Compound Type can group multiple values into one. Rust has the two primitive compound types tuples and arrays. Tuples have a fixed length and can store different types. Variables can either be accessed with pattern matching or with dot notation.

```
let tup = (500, 4.2, 'a');
let (x,y,z) = tup;
let five_hundred = tup.0;
```

Arrays have a fixed length and every element must have the same type and are written as a comma-separated list inside square brackets.

```
let a = [1, 2, 3];
let b = [3; 5]; // == [3, 3, 3, 3, 3]
let c: [i32; 5]; // [type; size]
c[0] = a[2];
```

In contrast to many low level languages Rust checks for *index out of bound* access at runtime to prevent invalid memory access.

4.3 Ownership

A uncommon feature in Rust is the ownership and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector. Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap, so you don't run out of space are all problems that ownership addresses.

Ownership is an approach where memory is managed through a set of rules which are checked at compile time. If any of these rules are violated, the program won't compile.

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

In this section the string Data type is used to illustrate the rules of ownership. Strings are more complex data type and have to be stored on the heap and therefore are a great example for ownership. Rust uses two types of stings, string literals who are hardcoded in the program and are immutable: `let s = "hello";`. These can easily be stored on the stack and be popped off when out of scope. The other string type have unknown size at compile time and hence have to be stored on the heap and can be mutated:

```
let mut s = String::from("hello");
s.push_str(", world!"); // appends a literal to a String.
println!("{}", s); // This will print `hello, world!`
```

This type of data must request memory from the memory allocator and needs a way to free this memory when it is no longer needed. In Rust memory is automatically freed once the variable that owns it goes out of scope. When a variable goes out of scope Rust calls the special function `drop` to free the memory.
When multiply variables use the same data the interaction with the memory becomes more complicated:

```
let s1 = String::from("hello");
let s2 = s1;
```

In this code snippet `s1` and `s2` point to the same memory and if and dropping both variables what would be freeing twice, what is known as double free bug, and can potentially lead to security vulnerabilities.
So Rust considers `s1` after the line `let s2 = s1;` no longer as valid. In addition to make a shallow copy of `s1` Rust also invalids the variable, this is known as a *move*.
Passing a value to a function works similar to assigning a value to a variable.
Passing a variable will move or copy it.
Returning values can also transfer ownership.
In conclusion ownership of a variable follows the following pattern: assigning a value to another variable moves it. When a variable that includes data on the heap goes out of scope, the value will be dropped unless ownership of the data has been moved to another variable. To avoid the need of returning ownerships of previously taken variables, Rust has a feature for using a value without transferring ownership, called *references*.

4.3.a References

While taking ownership and then returning it again works, the following code 4.3.1 snippet shows that it can be a bit tedious.

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Code 4.3.1: Returning ownership of parameters [15].

In this code the function has to return a tuple in order to return the ownership. So anything that gets passed to the function has to be passed back, if it is going to be used again. As well as the values that was going to be returned. This is to much hassle for a concept that should be common.

The issue with the code is that the string has to be returned so it can be used by the `print` function and therefore has to be returned with the value in a tuple.

Alternative a reference to the value can be provided. A reference, like a pointer, is a address to stored data, but that data is the address of data owned by some other variable, illustrated in fig. 3. Unlike a pointer, a reference is guarantied to to point to a valid value of a particular type for the life of that reference.

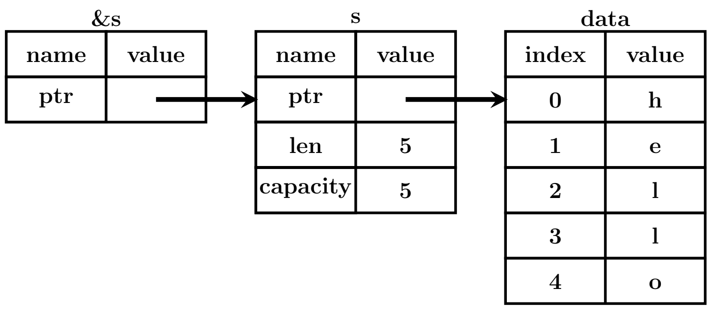

Fig. 3: A diagram of `&String: s` pointing at `String s`

Code 4.3.2 refactors 4.3.1 so the `calculate_length` function has a reference rather than the ownership of the object. Instead of capitalise on tuples the function now take a `&String` rather than a `String` as parameter.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Code 4.3.2: Borrow ownership of parameters [15].

The `&s` syntax allows to create a reference to a value without owning it, therefore the value it points to will not be dropped when the reference goes out of scope. Furthermore when functions have references as parameter instead of actual values, there is no need to return the value in order to pass the ownership

```
fn main() {
  let s = String::from("hello");
  change(&s);
}
fn change(some_string: &String) {
  some_string.push(", world");
}
```

Code 4.3.3: Attempt to modify the borrowed sting reference.

back since the function never had it.

This action of creating a reference is called *borrowing*. References are just like variables immutable by default, so modifying a borrowed value (like in Code 4.3.3) will lead to an error.

4.3.b Mutable References

To adjust the code 4.3.3 so that it allows to modify the borrowed value a few small tweaks are necessary. With the prefix `&mut` s becomes a mutable reference, the signature of the function is also change so it accepts a mutable String reference (see code 4.3.4).

```
fn main() {
  let mut s = String::from("hello");

  change(&mut s);
}

fn change(some_string: &mut String) {
  some_string.push_str(", world";
}
```

Code 4.3.4: Modify the borrowed sting reference of a mutable reference.

Mutable references have the restriction that there may not be more than one mutable reference to the same value in the same scope at a time. That is why the following code snippet would result in an error:

```rust
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;
println!("{}, {}", r1, r2};
```

The benefit of this restriction is that Rust can prevent data races at compile time. Data races can cause undefined behavior and can be difficult to fix, therefore Rust prevents this problem already at compile time. Rust enforces a similar rule for combining of mutable and immutable references:

```rust
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

Rust allows at any given time to have either one mutable reference or any number of immutable references in the same scope. Note that the scope of a reference starts from its introduction and continues till the last time that reference is used. Hence the following code is valid:

```rust
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{}, {}, and {}", r1, r2);
// r1 and r2 will not be used after this line

let r3 = &mut s; // no problem
println!("{r3}");
```

With this feature the Rust compiler indicates a potential bug early and points out exactly where the problem is. For example does this feature prevents dangling pointers - a pointer that do not resolve to a valid destination - by design: The compiler ensures that the data will not go out of scope before the reference does.

5 Implementation of Witness Extraction and Validation

For better performance witness extraction and emulation has to be enabled with a new flag (–witness, -w). With no witness flag the witness is still extracted, only the printing (section 5.3) and the emulation (section 5.4) is disabled. This is because of the low performance loss and the distributed code of the extraction.

Unicorn generates a Gate model representing the program. To solve the gate model, it is converted into a CNF. In this process every gate has to be visited by the CNF builder, this includes the input-gates. When a input-gate is visited the gate is recorded with its key (variable *name*) and reference. After the SAT-solver solved the CNF and the CNF is satisfiability, the recorded *names* are used to map the assignments of the CNF formula to the corresponding input-gates and save in the witness. After that the witness is printed, emulated and the program continues with the next bad state.

5.1 Main Implementation

The `main` function of Unicorn invokes the `solve_bad_states` function 5.1.1 located in the `sat_solver.rs` crate, see appendix B.3 for the full code.
This function takes several parameters and pass them forward to the `process_all_bad_states` function of the corresponding SAT solver and returns `Result<()>`. `Result` is an enum, which is a type that can be in one of multiple possible states. Result's variants are `Ok` and `Err`. The Ok variant indicates a successful operation, often containing the generated value. The Err variant means the operation failed, and Err contains information about how or why the operation failed.

```
pub fn solve_bad_states(
    gate_model: &GateModel,
    sat_type: SatType,
    terminate_on_bad: bool,
    one_query: bool,
    emulator: EmulatorState,
    extract_witness: bool,
) -> Result<()>
```

Code 5.1.1: The function solve_bad_states process bad states using different solvers based on the sat_type parameter.

The parameters include `GateModel`, representing the evaluated program, `SatType` determines the used sat solver. The boolean `terminate_on_bad` forces Unicorn to terminate at the first found bad state. `EmulatorState` is used by the emulator for validation.
The boolean `extract_witness` enables the witness extraction and validation. At this point it should be mentioned that the witness extraction and validation are currently only implemented for the Kissat SAT solver, as the implementation in this thesis serves as proof of concept. However, due to the generic nature of the implementation, extending it to the other solvers should be a straightforward task. Furthermore, `one_query == 1` is not supported by the witness extraction.
In `process_all_bad_states` (see code B.3.1 line 222) an iterator `zip` is created that pairs `bad_state` and `gate` from the `gate_model` and calls for every pair `(bad_state, gate)` the function 5.1.2 `process_single_bad_sate`.

```
fn process_single_bad_state<S: SATSolver>(
  solver: &mut S,
  gate_model: &GateModel,
  bad_state_: Option<&NodeRef>,
  gate: &GateRef,
  terminate_on_bad: bool,
  one_query: bool,
  (emulator, extract_witness): (&mut EmulatorState, bool),
) -> Result<()>
```

Code 5.1.2: The function solve_single_bad_state process a single bad states with the corresponding SAT solver.

In the `process_single_bad_state` function, the following functions are called: `solver.decide` (section 5.2), which returns the SATSolution enum (see code 5.1.3) containing the witness; `print_witness()` (section 5.3); and `emulate_witness()` (section 5.4). These functions represent the primary implementations for this thesis.

```
enum SATSolution {
    Sat(Option<Witness>),
    Unsat,
    Timeout,
}
```

Code 5.1.3: This enum is returned by the decide function in the SAT solvers and implies the satisfiability of a bad state.

5.2 Decide Function

The decide function, implemented for every supported SAT solver, is located in the `sat_solver.rs` file. In the decide function of the Kissat implementation, a CNF builder is created, and the bad state under investigation in this loop is added (lines 4 and 5 in Code 5.2.1). The CNF is then created by visiting all nodes, starting at the bad state gate, and recursively traversing through the model.

```
1 fn decide(&mut self, gate_model: &GateModel, gate: &GateRef) -> SATSolution {
2   let mut builder = CNFBuilder::<KissatContainer>::new();
3
4   let bad_state_var = builder.visit(gate);
5   builder.container_mut().add_clause(&[bad_state_var]);
```

Code 5.2.1: Part of the decide function for the 'kissat' solver that creates the CNF.

In the subsequent lines (see Code 5.2.2), the constraints are added to the CNF. These constraints are necessary for enforcing the equation $\text{dividend} == \text{divisor} * \text{quotient} + \text{remainder}$ for division and remainder operations.

```
  for (gate, val) in &gate_model.constraints {
    let constraint_var = builder.visit(&gate.value);
    let constraint_lit = if *val {
      KissatContainer::var(constraint_var)
    } else {
      KissatContainer::neg(constraint_var)
    };
    builder.container_mut().add_clause(&[constraint_lit]);
  }
```

Code 5.2.2: Adds the constraints to the CNF.

```
    (...)
        Gate::InputBit { name } => {
        let gate_var = self.next_var();
        self.record_variable_name(gate_var, name.clone());
        self.record_input(gate_var, gate);
        gate_var
      }
```

Code 5.2.3: The gate that represents a bit input.

During the visitation of all gates, the algorithm also visits the input gates. It is then possible to map the input gates (see Code 5.2.3) with the corresponding variable literal for later use (see Code 5.2.4). Line 15 in Code 5.2.8 creates a

```
fn record_input(&mut self, var: Self::Variable, gate: &GateRef) {
  if let Gate::InputBit { name } = &*gate.borrow() {
    if name.len() > 1 && name[1..].starts_with("-byte-input") {
      self.variables.insert(var, gate.clone());
    }
  }
}
```

Code 5.2.4: The function in the kissat container that records the inputs.

mutable `KissatContainer` variable, named `cnf`, by calling the `container_mut()` method from the builder object.

In the next line, the ownership of the field `state` in `cnf` is taken and stored in the variable `state`. The `take()` method extracts the value from the Option and returns it, leaving a None in its place. The `unwrap()` method extracts the value from the Option type and panics if, for any reason, the state is None. The `keys()` function, in line 17, returns an iterator over the keys of the `variables` fields in the `cnf` container.

Finally, in the line 19, Kissat is used to solve the CNF by calling the `solve()` function of the Kissat wrapper. The solver returns the enum 5.2.6. If the solver returns `AnyState::UNSAT`, the `SATSolution::Unsat` is returned and then continues with the next bad state. For the case `AnyState::INPUT`, a panic is raised as SAT or UNSAT is expected. If the solver returns `AnyState::SAT`, the solver has found an assignment such that the bad state evaluates to TRUE. This assignment is provided inside the Option as a `SATState` struct from the Kissat wrapper (see Code 5.2.5).

In the `AnyState::SAT` branch of the matcher, the newly implemented struct `Witness` (see Code 5.2.7) is created with some initial values. This new struct

```
pub struct SATState {
    _internal: PhantomData<()>,
}
```

Code 5.2.5: SATState is the state type which encodes that the solver has found the given formula to be satisfiable.

```
enum SATSolution {
  Sat(Option<Witness>),
  Unsat,
  Timeout,
}
```

Code 5.2.6: Enum for the return value of the SAT solvers.

is used so that any SAT solver can utilize the witness extraction and emulation. The struct contains the name and the reference to the `bad_state_gate`, that is satisfiable, the `gate_assignment`, a hashmap that maps the `InputBit` gate reference to the corresponding assignment. The `input_bytes` is used as a stack that contains the byte that actually needs to be inputted in the runtime and is filled later.

```
pub struct Witness {
  pub name: String,
  pub bad_state_gate: GateRef,
  pub gate_assignment: HashMap<HashableGateRef, bool>,
  pub input_bytes: Vec<usize>,
}
```

Code 5.2.7: Struct that contains all information needed for the witness extraction.

Then, the code iterates over the `literals` and updates the `witness.gate_assignment` map based on the solver's value in the code block from lines 28 to 40. Eventually, the witness is returned as an option in `SATSolution::Sat(Some(witness))` 5.2.6.

```
  let cnf = builder.container_mut();
  let state = cnf.state.take().unwrap();
  let literals = cnf.variables.keys();

  match cnf.solver.solve(state).unwrap() {
    AnyState::SAT(sat_state) => {
      let mut witness = Witness {
        name: String::new(),
        bad_state_gate: gate.clone(),
        gate_assignment: HashMap::new(),
        input_bytes: vec![],
      };
      let mut sat = sat_state;
      for literal in literals.copied() {
        let value = cnf.solver.value(literal, sat).unwrap();
        let assignment = match value.0 {
          Assignment::True => true,
          Assignment::False => false,
          Assignment::Both => false,
        };
        sat = value.1;
        let gate_ref = cnf.variables.get(&literal).cloned();
        witness
          .gate_assignment
          .insert(HashableGateRef::from(gate_ref.unwrap()), assignment);
      }
      SATSolution::Sat(Some(witness))
    }
    AnyState::UNSAT(..) => SATSolution::Unsat,
    AnyState::INPUT(..) => panic!("expecting 'SAT' or 'UNSAT' here"),
  }
}
```

Code 5.2.8: This code snippet shows the final part of the decide function, where the SAT solver is used to solve the CNF. Depending on the solver's return value, the function either returns an Unsat solution, panics, or constructs a Witness object with the satisfying assignment and returns it as part of a Sat solution.

5.3 Print Witness

The `SATSolution` is returned to the `process_single_bad_state()` function (see line 150 code B.3.1). If `SATSolution` equals `SAT(witness)` and the `extract_witness` flag is true the code branches into the `S::print_witness()` method (see code 5.3.1).
In the code 5.3.2, the reference to the `HashMap`, which filled in the decide function (see code 5.2.8), is stored in the variable `witness`. Additionally, the variables `input` and `start` are initialized. The code also affirms that the witness is not empty. Next, the code enters the first of two for loops, iterating over all Gate references using the iterator from `witness.keys()`.

```
let solution = solver.decide(gate_model, gate);
match solution {
  SATSolution::Sat(witness_opt) => {
    (...)
    if extract_witness {
      match witness_opt {
        Some(mut witness) => {
          witness.name = name.clone().unwrap();
          println!("solution by {}:", S::name());
          S::print_witness(&mut witness);
          emulate_witness(emulator, witness);
        }
```

Code 5.3.1: In this code the name of the bad state is stored in the `witness` and the methods `print` and `emulate` are invoked.

```
  fn print_witness(witness_ref: &mut Witness) {
    let witness = &witness_ref.gate_assignment;
    let mut input: HashMap<u64, Vec<bool>> = HashMap::new();
    let mut start;
    if witness.is_empty() {
      println!("No witness produced for this bad state.");
      return;
    }
```

Code 5.3.2: Declaration of the variables.

To ensure code safety, `if let` is used as pattern matching against the `InputBit` gate for the keys of `witness`, while also extracting the `name` field (see code 5.3.3).

```
    for key in witness.keys() {
      if let Gate::InputBit { name } = key.value.borrow().deref() {
```

Code 5.3.3: For loop head and pattern matching.

The Information (line number and bit number) are stored in the string `name` of the input-Gates.
Name has the pattern `"1-byte-input[n=\d][bit=\d]"`, the code 5.3.4 extract these values and change the `bit` to most significant bit format, in dependence of the input gate byte size.

In the code block 5.3.5, the bits in the byte input are assigned and stored in a dynamically growing vector within the `input` `HashMap`. The logic ensures that if a bit value already exists, it is updated with the new assignment. Otherwise, a new entry is created in the HashMap to store the bit value. This approach becomes necessary because the `Gate::InputBit` instances are not explicitly stored, allowing the bits to arrive in any order.

```
        // name = "1-byte-input[n=\d][bit=\d]"
        //                        ^--- the 15th char
        // assert: bit is explicit
        start = name.find(']').unwrap();
        let n = name[15..start].parse().unwrap();
        let size = name.chars().next().and_then(|c| c.to_digit(10)).unwrap() as usize;
        let mut bit: usize = name[start + 6..name.len() - 1].parse().unwrap();
        bit = size * 8 - 1 - bit;
```

Code 5.3.4: This code shows how the names of the byte inputs are parsed.

```
        let bit_assignment = *witness.get(key).unwrap();
        if let Some(bits) = input.get_mut(&n) {
          if let Some(bit_value) = bits.get_mut(bit) {
            *bit_value = bit_assignment;
          } else {
            bits.resize(bit + 1, bit_assignment);
          }
        } else {
          let mut bits: Vec<bool> = Vec::new();
          bits.resize(bit + 1, bit_assignment);
          input.insert(n, bits);
        }
      } else {
        panic!("Gates in the Witness must be Input Bit Gates.");
      }
    }
```

Code 5.3.5: In this code snippet the input HashMap is updated.

The code snippet, shown in code 5.3.6, first constructs an output string with a string builder. This string represents a binary sequence received from the `bits` vector in `inputs`. Next, the code converts this binary string to an unsigned integer (`usize`). Finally, the resulting integer value is stored in the `input_bytes` field within the `Witness` struct. This storage is later used by the emulator to verify the witness.

```
    for bits in input {
      let mut output = String::new();
      print!("input at [n={}] ", bits.0);
      for bit in bits.1 {
        output.push(if bit { '1' } else { '0' });
      }
      let bits_as_int = usize::from_str_radix(&output, 2).unwrap();
      witness_ref.input_bytes.push(bits_as_int);
      println!("{} {}", output, bits_as_int);
    }
  }
}
```

Code 5.3.6: This code snippet eventually prints the witness and the line where the original code prompts for user input.

5.4 Emulate Inputs

To emulate the witness, the emulator had to be adapted so that the system call `read` B.1.1 is bypassed. Instead of reading from normal input prompts, the operator `read` now reads from memory, but it should still function normally under typical circumstances. This adaptation is achieved by using a vector as a stack for the inputs.

```
1  fn syscall_read(state: &mut EmulatorState) {
2    let fd = state.get_reg(Register::A0);
3    let buffer = state.get_reg(Register::A1);
4    let size = state.get_reg(Register::A2);
5
6    if fd == 0 && !state.std_inputs.is_empty() {
7      for adr in (buffer..buffer + size).step_by(riscu::WORD_SIZE) {
8        let byte = state.std_inputs.pop().unwrap();
9        state.set_mem(adr, byte as EmulatorValue);
10     }
11   } else {
12     // Check provided address is valid, iterate through the buffer word
13     // by word, and emulate `read` system call via `std::io::Read`.
```

Code 5.4.1: The system call read located in the emulator file.

If the stack is empty, the read call behaves as usual; otherwise, it reads from the top of the stack. For this the code 5.4.1 checks in line 6 checks if `fd == 0`, so that file direction is the standard input sys_in and the input stack in the `EulatorState` is not empty, else the read call progresses as usually.
In this *if* branch `size` many bytes are pop out of the stack and stored in the memory at address `buffer + i * WORD_SIZE`. In this stack, the assignments of the witness is placed, but any `usize` value can be added to this vector via getter and setter methods (see code 5.4.3). As a byproduct, this approach allows passing inputs to the emulator via the attributes of the flag `--stdin`. However, emulating witnesses introduces another issue: witnesses in bad states are undesirable and can trigger a panic, causing the emulator to terminate. To handle this, a catch unwind block 5.4.2 is necessary.
Additionally, as references can't be transferred easily across the catch unwind boundary the stdin field is set in the original `EmulatorState` instead of the clone. More important the `EmulatorState` must be cloneable, so the `EmulatorState` now includes a `clone` function B.2.2.

```rust
fn emulate_witness(emulator: &mut EmulatorState, witness: Witness) {
  (...)
  emulator.set_stdin(witness.input_bytes);
  let result = panic::catch_unwind(|| {
    let mut emulator_clone: EmulatorState;
    emulator_clone = emulator.clone();
    emulator_clone.run();
  });
  if result.is_ok() {
    println!("Bad state {} did not produce expected panic.", witness.name);
  }
}
```

Code 5.4.2: This code shows how the witness is emulated inside a catch unwind block.

```rust
// The emulator reads from this vector instead of the Stdin when the
// read syscall (with file direction 0) is called in the emulated code.
// Will call the syscall again as soon the std_inputs is empty.
pub fn set_stdin(&mut self, inputs: Vec<usize>) {
  self.std_inputs = inputs;
}
pub fn get_stdin(&self) -> &Vec<usize> {
  &self.std_inputs
}
```

Code 5.4.3: Setter and getter methods for the standard input stack.

6 Examples

This section the previous workflow and the workflow with the witness extraction implementations is presented. Additionally some examples are granted. In the first example the code contains a division by zero, what

```c
uint64_t main() {
  uint64_t  a;
  uint64_t* x;

  x = malloc(8);
  *x = 0;

  read(0, x, 1);

  *x = *x - 48;

  // division by zero if the input was '0' (== 48)
  a = 41 + (1 / *x);

  // division by zero if the input was '2' (== 50)
  if (*x == 2)
    a = 41 + (1 / 0);

  if (a == 42)
    return 1;
  else
    return 0;
}
```

Code 6.0.1: Example C Code for the bad state: 'divison-by-zero' obviously is bad. The code 6.0.1 can run at two positions in an error, first in line 13 where it use the input as divisor and secondly in the if branch where it divides by zero if the input was the char '2'. Initially a binary of the code must be generated, selfie [7] can be used as compiler but any RISC-V binary works.

```
$ ./selfie -c examples/division.c -o division.m
```

Than unicorn is used to produce a unrolled BTOR2 file and the bad states are printed to get the depth where the bad state occurs.

```
$ ./unicorn beator division.m -u 100 -o division-unrolled.btor2
$ cat division-unrolled.btor2 | grep bad
```

This output includes among other information the bad state division by zero and the instruction number $n = 98$.

```
10001877 bad 10001876 division-by-zero[n=98]
10001883 bad 10001882 memory-access-above-stack[n=95]
10001889 bad 10001888 memory-access-between-dyn-and-max-stack[n=95]
...
10001912 bad 10001911 division-by-zero[n=85]
10001975 bad 10001974 memory-access-below-data
10001980 bad 10001979 memory-access-between-data-and-heap
....
```

With the depth we can use the SMT solver to efficiently get the satisfiability of the bad states.

```
$ ./unicorn beator division.m -u 99 -p -s boolector
Bad state 'division-by-zero[n=98]' is satisfiable!
Bad state 'division-by-zero[n=85]' is satisfiable!
```

Next unicorn is used to create a CNF file and passed to minisat [14] to get a file with the satisfiability and the assignment of all 1235 variables.

```
$ ./unicorn beator division.m -u 99 -p -b -d -o division.cnf
$ minisat division.cnf division.sat

$ cat division.sat
SAT
-1 -2 3 -4 5 6 -7 8 9 10 11 -12 13 -14 -15 16 -17 -18 19 -20 ...
```

To get the witness out of these data, a shell script (sat2human.sh) has to be used to map the inputs with the variables.

```
$ ./tools/sat2human.sh division.cnf division.sat
#: 0 1-byte-input[n=70][bit=7]
#: 0 1-byte-input[n=70][bit=6]
#: 1 1-byte-input[n=70][bit=5]
#: 1 1-byte-input[n=70][bit=4]
#92: 0 1-byte-input[n=70][bit=3]
#95: 0 1-byte-input[n=70][bit=2]
#98: 1 1-byte-input[n=70][bit=1]
#01: 0 1-byte-input[n=70][bit=0]
#99: 1 division-by-zero[n=98]
#61: 0 memory-access-above-stack[n=95]
#007: 0 memory-access-between-dyn-and-max-stack[n=95]
#045: 0 memory-access-between-heap-and-stack[n=95]
#069: 0 memory-access-between-max-and-dyn-heap[n=95]
#107: 0 memory-access-between-data-and-heap[n=95]
#108: 0 memory-access-below-data[n=95]
#235: 0 division-by-zero[n=85]
```

This output contains the information that $00110010 = 50$ as input leads to a division by zero at depth 98 but nothing about the second bad state at depth 85.

The same process can be carried out with the new witness extraction more easily. The depth is extracted the same way, with the unrolled BTOR2 file. Kissat without the witness flag prints that division by zero is satisfiable at $n = 98$ and $n = 85$.

```
$ ./unicorn beator division.m -u 99 -p -b --sat-solver kissat
Bad state 'division-by-zero[n=98]' is satisfiable (Kissat)!
Bad state 'division-by-zero[n=85]' is satisfiable (Kissat)!
```

With the witness extraction enabled (-w) the following output is printed:

```
$ ./unicorn beator division.m -u 99 -p -b --sat-solver kissat -w
Bad state 'division-by-zero[n=98]' is satisfiable (Kissat)!
Solution by Kissat: input at [n=70] 00110010 50
Emulating bad state "division-by-zero[n=98]" with input [50]:
--------------------------------------------------
thread 'main' panicked at 'check for non-zero divisor', src/emulate.rs:875:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
--------------------------------------------------

Bad state 'division-by-zero[n=85]' is satisfiable (Kissat)!
Solution by Kissat: input at [n=70] 00110000 48
Emulating bad state "division-by-zero[n=85]" with input [48]:
--------------------------------------------------
thread 'main' panicked at 'check for non-zero divisor', src/emulate.rs:875:5
--------------------------------------------------
```

The output presents the bad state that is satisfiable, the witness as binary and integer whit the input line, and than emulates the input with the corresponding error message. This is printed for all witnesses not only the first one.

6.1 Example: 8 Byte input

This example shows that the witness extraction can extract and emulate eight byte inputs. The output below shows the correct witness and the emulator

```c
uint64_t main() {
  uint64_t a;
  uint64_t* x;

  x = malloc(8);
  *x = 0;

  read(0, x, 8);

  a = 3544668469065756977;
  // input == "11111111" ==  3544668469065756977
  if (*x == a)
    *x = 42 / 0;

  return 0;
}
```

Code 6.1.1: Example C Code for eight byte input.

panics because of the division by zero.

```
$ ./unicorn beator eigt-byte-input.m -u 90 -p -b --sat-solver kissat -w
Bad state 'division-by-zero[n=87]' is satisfiable (Kissat)!
Solution by Kissat:
input at [n=70] 0011000100110001001100010011000100110001001100010011000100110001
↪  3544668469065756977
Emulating bad state "division-by-zero[n=87]" with input [3544668469065756977]:
--------------------------------------------------
thread 'main' panicked at 'check for non-zero divisor', src/emulate.rs:875:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
--------------------------------------------------
```

6.2 Example: Invalid memory access

The last example is about invalid memory access, first, with the input 49 = '1', the code tries to access memory above the virtual address space. With the input 50 = '2' the code access memory below the bump pointer, so between the data and heap segment, what is also not allowed.

```c
uint64_t main() {
  uint64_t  a;
  uint64_t* x;

  x = malloc(8);
  *x = 0;

  read(0, x, 1);

  if (*x == 49)
    // address outside virtual address space -> invalid memory access
    *(x + (4 * 1024 * 1024 * 1024)) = 0;
  if (*x == 50)
    //  address between data and heap -> invalid memory access
    *(x + -2) = 0;

  return 0;
}
```

Code 6.2.1: Example C Code for invalid memory access.

The boolector correctly identifies both bad states 6.2.2 but the emulator doesn't throw the expected error (see highlighted line in 6.2.3), what indicates an error in the emulator of unicorn.

```
$ ./unicorn beator memory-access.m -u 120 -p -b -s boolector
Bad state 'memory-access-between-data-and-heap[n=96]' is satisfiable!
Bad state 'memory-access-above-stack[n=94]' is satisfiable!
```

Code 6.2.2: Output of the boolector for the code 6.2.1.

Selfie itself will throw an segmentation fault when execute the code 6.2.1 with ′2′ as input.

```
./selfie: selfie executing 64-bit RISC-U binary memory-access.m with 128MB physical memory
↪  on 64-bit mipster
./selfie: >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
2
./selfie: context memory-access.m threw uncaught exception: segmentation fault at address
↪  0x00011FF0./selfie: selfie terminating 64-bit RISC-U binary memory-access.m with exit
↪  code 27
./selfie: --------------------------------------------------------------------------------
```

```
$ ./unicorn beator memory-access.m -u 99 -p -b --sat-solver kissat -w
Bad state 'memory-access-between-data-and-heap[n=96]' is satisfiable (Kissat)!
Solution by Kissat:
input at [n=70] 00110010 50
Emulating bad state "memory-access-between-data-and-heap[n=96]" with input [50]:
--------------------------------------------------

program exiting with exit code 0
Bad state memory-access-between-data-and-heap[n=96] did not produce expected panic.
--------------------------------------------------

Bad state 'memory-access-above-stack[n=94]' is satisfiable (Kissat)!
Solution by Kissat:
input at [n=70] 00110001 49
Emulating bad state "memory-access-above-stack[n=94]" with input [49]:
--------------------------------------------------
thread 'main' panicked at 'range start index 34359812096 out of range for slice of length
↪  1048576', src/emulate.rs:174:38
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
--------------------------------------------------
```

Code 6.2.3: Output of kissat for the code 6.2.1.

7 Conclusion

This work surveyed the possibility to extend Unicorn with a witness extraction and validation functionality. The implementation successfully set up this expansion and was able to spot some errors both in the emulator and Unicorn. Notable, the bad state 'no zero exit code' is not consistently recognised by the sat solver.

The key contribution of this work was to demonstrate the possibility of the functionality. By focusing on Kissat, one of the most powerful SAT solver supported by Unicorn, the implementation becomes immediately usable. Furthermore, transitioning to the other solvers is relatively straightforward. However, it's essential to acknowledge the limitations of witness validation. Specifically, the approach- how the inputs are bypassed in the read syscall- may not be suitable for all C programs. The issue arises from how witness inputs are interpreted—specifically, as the first n inputs. Consequently, correct validation becomes impossible if there are inputs independent of the Bad State that are not read in at the end. Addressing this restriction would require significant effort, but can be easily bypassed.

In summary, this implementation demonstrates a proof of concept for the witness extraction and validation in Unicorn, while having some flaws it could also identify some errors in Unicorn.

8 References

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 50.

[2] Miquel Bofill et al. "A Write-Based Solver for SAT Modulo the Theory of Arrays". In: *2008 Formal Methods in Computer-Aided Design*. 2008, pp. 1–8. DOI: 10.1109/FMCAD.2008.ECP.18.

[3] Roberto Bruttomesso. "RTL Verification: From SAT to SMT (BV)". PhD thesis. PhD thesis, University of Trento, 2008.

[4] ABKFM Fleury and Maximilian Heisinger. "Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020". In: *SAT COMPETITION* 2020 (2020), p. 50.

[5] Nils Froleyks et al. "SAT Competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103572. URL: https://www.sciencedirect.com/science/article/pii/S0004370221001235.

[6] Jannis Harder. "Varisat, a SAT Solver Written in Rust". In: *SAT COMPETITION 2018* (), p. 49.

[7] Christoph Kirsch. *Selfie: An educational software system of a tiny self-compiling C compiler, a tiny self-executing RISC-V emulator, and a tiny self-hosting RISC-V hypervisor.* https://github.com/cksystemsteaching/selfie. Computer Science for All courses, University of Salzburg, Austria. 2014.

[8] Christoph Kirsch, Daniel Kocher, and Michael Starzinger. *Unicorn*. English. 2022.

[9] Christoph M Kirsch and Stefanie Muroya Lei. "Quantum Advantage for All". In: *arXiv preprint arXiv:2111.12063* (2021).

[10] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[11] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[12] Aina Niemetz et al. "Btor2, BtorMC and Boolector 3.0". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 587–595.

[13] Richard J. Orgass. “J. McCarthy. Towards a mathematical science of computation.” In: *Journal of Symbolic Logic* 36.2 (1971), pp. 346–347. DOI: 10.2307/2270319.

[14] Niklas Sörensson. “Minisat 2.2 and minisat++ 1.1”. In: *A short description in SAT Race* 2010 (2010).

[15] Rust Team. *Rust Documentation*. 2024. URL: https://doc.rust-lang.org/.

[16] R. Winter. *Theoretische Informatik: Grundlagen mit Übungsaufgaben und Lösungen*. Reprint 2015. Oldenbourg Wissenschaftsverlag, 2009, pp. 165–166.

Appendices

A List of Bad States

If the program enters a undesired state or deviate from expected behavior we referred to it as bad state. Those can lead to errors, crashes, security vulnerabilities, or incorrect results. The table 2 lists all bad states that Unicorn is able to detected.

bad states
invalid-syscall-id
memory-access-below-data
memory-access-between-data-and-heap
memory-access-between-max-and-dyn-heap
memory-access-between-heap-and-stack
memory-access-between-dyn-and-max-heap
memory-access-above-stack
division-by-zero
remainder-by-zero
non-zero-exit-code

Table 2: List of the bad states in Unicorn.

B Unicorn Code

B.1 System Call Read

Code B.1.1: The system call read located in the emulator file.

```
fn syscall_read(state: &mut EmulatorState) {
  let fd = state.get_reg(Register::A0);
  let buffer = state.get_reg(Register::A1);
  let size = state.get_reg(Register::A2);

  if fd == 0 && !state.std_inputs.is_empty() {
    for adr in (buffer..buffer + size).step_by(riscu::WORD_SIZE) {
      let byte = state.std_inputs.pop().unwrap();
      state.set_mem(adr, byte as EmulatorValue);
    }
  } else {
    // Check provided address is valid, iterate through the buffer word
    // by word, and emulate `read` system call via `std::io::Read`.
    assert!(buffer & WORD_SIZE_MASK == 0, "buffer pointer aligned");
    let mut total_bytes = 0; // counts total bytes read
    let mut tmp_buffer: Vec<u8> = vec![0; 8]; // scratch buffer
    for adr in (buffer..buffer + size).step_by(riscu::WORD_SIZE) {
    let bytes_to_read = min(size as usize - total_bytes, riscu::WORD_SIZE);
    LittleEndian::write_u64(&mut tmp_buffer, state.get_mem(adr));
    let bytes = &mut tmp_buffer[0..bytes_to_read]; // only for safety
    let bytes_read = state.fd_read(fd).read(bytes).expect("read success");
    state.set_mem(adr, LittleEndian::read_u64(&tmp_buffer));
    total_bytes += bytes_read; // tally all bytes
    if bytes_read != bytes_to_read {
      break;
    }
  }
  let result = total_bytes as u64;

  state.set_reg(Register::A0, result);
  debug!("read({},{:#},{}) -> {}", fd, buffer, size, result);
  }
}
```

B.2 Emulator State Clone

Code B.2.1: The EmulatorState struct representing the state of an emulator. It includes fields for registers, memory, program counter, program break, file handles, and I/O streams.

```
pub type EmulatorValue = u64;

#derive(Debug)]
pub struct EmulatorState {
  registers: Vec<EmulatorValue>,
  memory: Vec<u8>,
  program_counter: EmulatorValue,
  program_break: EmulatorValue,
  opened: Vec<File>,
  running: bool,
  stdin: Stdin,
  stdout: Stdout,
  std_inputs: Vec<usize>,
}
```

Code B.2.2: The EmulatorState struct implements the Clone trait, providing a custom `clone()` method. It creates a new EmulatorState instance and files the fields with new and cloned objects.

```
impl Clone for EmulatorState {
  fn clone(&self) -> Self {
    EmulatorState {
      registers: self.registers.clone(),
      memory: self.memory.clone(),
      program_counter: self.program_counter,
      program_break: self.program_break,
      opened: Vec::new(),
      running: self.running,
      stdin: io::stdin(),
      stdout: io::stdout(),
      std_inputs: self.std_inputs.clone(),
    }
  }
}
```

Code B.2.3: The EmulatorState struct provides a constructor method `new()` that initializes a new instance of the emulator state. It sets default values for registers, memory, program counter, program break, and I/O streams.

```
impl EmulatorState {
  pub fn new(memory_size: usize) -> Self {
    Self {
      registers: vec![0; NUMBER_OF_REGISTERS],
      memory: vec![0; memory_size],
      program_counter: 0,
      program_break: 0,
      opened: Vec::new(),
      running: false,
      stdin: io::stdin(),
      stdout: io::stdout(),
      std_inputs: Vec::new(),
    }
  }
```

B.3 Sat Solver

Code B.3.1: sat_solver.rs without the solvers 'varisat' and 'cadical'.

```
use crate::unicorn::bitblasting::{get_constant, or_gate, Gate,
GateModel, GateRef, Witness};
use crate::unicorn::{Node, NodeRef};
use crate::SatType;
use anyhow::{anyhow, Result};
use kissat_rs::Assignment;
use log::{debug, warn};
use std::collections::HashMap;
use std::ops::Deref;
use std::panic;
use unicorn::emulate::EmulatorState;

//
// Public Interface
//
#allow(unused_variables)]
pub fn solve_bad_states(
  gate_model: &GateModel,
  sat_type: SatType,
  terminate_on_bad: bool,
  one_query: bool,
  emulator: EmulatorState,
  extract_witness: bool,
) -> Result<()> {
  match sat_type {
    SatType::None => unreachable!(),
    #cfg(feature = "kissat")]
    SatType::Kissat =>
    process_all_bad_states::<kissat_impl::KissatSolver>(
      gate_model,
```

```
      terminate_on_bad,
      one_query,
      emulator,
      extract_witness,
    ),
    #cfg(feature = "varisat")]
    SatType::Varisat =>
    process_all_bad_states::<varisat_impl::VarisatSolver>(
      gate_model,
      terminate_on_bad,
      one_query,
      emulator,
      extract_witness,
    ),
    #cfg(feature = "cadical")]
    SatType::Cadical =>
    process_all_bad_states::<cadical_impl::CadicalSolver>(
      gate_model,
      terminate_on_bad,
      one_query,
      emulator,
      extract_witness,
    ),
  }
}

//
// Private Implementation
//

#should_panic]
fn emulate_witness(emulator: &mut EmulatorState, witness: Witness) {
  println!(
    "Emulating bad state {:?} with input {:?}:",
    witness.name,
    witness.input_bytes.clone()
  );
  println!("---------------------------------------------------");
  emulator.set_stdin(witness.input_bytes);
  let result = panic::catch_unwind(|| {
    let mut emulator_clone: EmulatorState;
    emulator_clone = emulator.clone();
    emulator_clone.run();
  });
  if result.is_ok() {
    println!("Bad state {} did not produce expected panic.", witness.name);
  }
  println!("---------------------------------------------------\n");
}

#allow(dead_code)]
#derive(Debug, Eq, PartialEq)]
enum SATSolution {
  Sat(Option<Witness>),
```

```
  Unsat,
  Timeout,
}

trait SATSolver {
  fn new() -> Self;
  fn name() -> &'static str;
  fn prepare(&mut self, gate_model: &GateModel);
  fn decide(&mut self, gate_model: &GateModel, gate: &GateRef) -> SATSolution;

  fn print_witness(witness_ref: &mut Witness) {
    let witness = &witness_ref.gate_assignment;
    let mut input: HashMap<u64, Vec<bool>> = HashMap::new();
    let mut start;
    if witness.is_empty() {
      panic!("Witness is empty");
    }

    for key in witness.keys() {
      if let Gate::InputBit { name } = key.value.borrow().deref() {
        // name = "1-byte-input[n=\d][bit=\d]"
        //            ^--- the 15th char
        // assert: bit is explicit
        start = name.find(']').unwrap();
        let n = name[15..start].parse().unwrap();

        let mut bit: usize = name[start + 6..name.len() - 1].parse().unwrap();
        bit = 7 - bit;

        input.entry(n).or_insert(Vec::new());

        let bit_assignment = match witness.get(key).unwrap() {
          Assignment::True => true,
          Assignment::False => false,
          Assignment::Both => false,
        };

        if let Some(bits) = input.get_mut(&n) {
          if let Some(bit_value) = bits.get_mut(bit) {
            *bit_value = bit_assignment;
          } else {
            bits.resize(bit + 1, bit_assignment);
          }
        } else {
          let mut bits: Vec<bool> = Vec::new();
          bits.resize(bit + 1, bit_assignment);
          input.insert(n, bits);
        }
      } else {
        panic!("Gates in the Witness must be Input Bit Gates.");
      }
    }
    for bits in input {
      let mut output = String::new();
```

```
      print!("input at [n={}] ", bits.0);
      for bit in bits.1 {
        output.push(if bit { '1' } else { '0' });
      }
      let bits_as_int = usize::from_str_radix(&output, 2).unwrap();
      witness_ref.input_bytes.push(bits_as_int);
      println!("{} {}", output, bits_as_int);
    }
  }
}

fn process_single_bad_state<S: SATSolver>(
  solver: &mut S,
  gate_model: &GateModel,
  bad_state_: Option<&NodeRef>,
  gate: &GateRef,
  terminate_on_bad: bool,
  one_query: bool,
  (emulator, extract_witness): (&mut EmulatorState, bool),
) -> Result<()> {
  if !one_query {
    let bad_state = bad_state_.unwrap();
    if let Node::Bad { name, .. } = &*bad_state.borrow() {
      println!(
        "process_single_bad_state {}",
        name.as_deref().unwrap_or("?")
      );
      let solution = solver.decide(gate_model, gate);
      match solution {
        SATSolution::Sat(witness_opt) => {
          warn!(
            "Bad state '{}' is satisfiable ({})!",
            name.as_deref().unwrap_or("?"),
            S::name()
          );
          if extract_witness {
            match witness_opt {
              Some(mut witness) => {
                witness.name = name.clone().unwrap();
                println!("solution by {}:", S::name());
                S::print_witness(&mut witness);
                emulate_witness(emulator, witness);
              }
              None => {
                println!("No Witness");
              }
            }
          }

          if terminate_on_bad {
            return Err(anyhow!("Bad state satisfiable"));
          }
        }
        SATSolution::Unsat => {
```

```
          debug!(
            "Bad state '{}' is unsatisfiable ({}).",
            name.as_deref().unwrap_or("?"),
            S::name()
          );
        }
        SATSolution::Timeout => unimplemented!(),
      }
      Ok(())
    } else {
      panic!("expecting 'Bad' node here");
    }
  } else {
    assert!(bad_state_.is_none());
    let solution = solver.decide(gate_model, gate);
    match solution {
      SATSolution::Sat(..) => {
        warn!("At least one bad state evaluates to true ({})", S::name());
      }
      SATSolution::Unsat => {
        debug!("No bad states occur ({}).", S::name());
      }
      SATSolution::Timeout => unimplemented!(),
    }
    Ok(())
  }
}

#allow(dead_code)]
fn process_all_bad_states<S: SATSolver>(
  gate_model: &GateModel,
  terminate_on_bad: bool,
  one_query: bool,
  mut emulator: EmulatorState,
  extract_witness: bool,
) -> Result<()> {
  debug!("Using {:?} to decide bad states ...", S::name());
  let mut solver = S::new();

  if !one_query {
    let zip = gate_model
      .bad_state_nodes
      .iter()
      .zip(gate_model.bad_state_gates.iter());

    for (bad_state, gate) in zip {
      process_single_bad_state(
        &mut solver,
        gate_model,
        Some(bad_state),
        gate,
        terminate_on_bad,
        one_query,
        (&mut emulator, extract_witness),
```

```
      )?;
    }
  } else {
    let mut ored_bad_states: GateRef;
    if gate_model.bad_state_gates.is_empty() {
      ored_bad_states = GateRef::from(Gate::ConstFalse);
    } else if gate_model.bad_state_gates.len() == 1 {
      ored_bad_states = gate_model.bad_state_gates[0].clone();
    } else {
      let first_element = gate_model.bad_state_gates[0].clone();
      let second_element = gate_model.bad_state_gates[1].clone();
      ored_bad_states = or_gate(
        get_constant(&first_element),
        get_constant(&second_element),
        &first_element,
        &second_element,
      );
    }
    for gate in gate_model.bad_state_gates.iter().skip(2) {
      ored_bad_states = or_gate(
        get_constant(&ored_bad_states),
        get_constant(gate),
        &ored_bad_states,
        gate,
      );
    }
    if let Some(value) = get_constant(&ored_bad_states) {
      if value {
        warn!("Bad state occurs");
      } else {
        warn!("No bad state occurs");
      }
    } else {
      process_single_bad_state(
        &mut solver,
        gate_model,
        None,
        &ored_bad_states,
        terminate_on_bad,
        one_query,
        (&mut emulator, extract_witness),
      )?;
    }
  }

  Ok(())
}

#cfg(feature = "kissat")]
pub mod kissat_impl {
  use crate::unicorn::bitblasting::{Gate, GateModel, GateRef, HashableGateRef, Witness};
  use crate::unicorn::cnf::{CNFBuilder, CNFContainer};
  use crate::unicorn::sat_solver::{SATSolution, SATSolver};
  use kissat_rs::{AnyState, Assignment, INPUTState, Literal, Solver};
```

```rust
use std::collections::HashMap;

pub struct KissatSolver {}

struct KissatContainer {
  current_var: i32,
  solver: Solver,
  state: Option<INPUTState>,
  variables: HashMap<Literal, GateRef>,
}

impl CNFContainer for KissatContainer {
  type Variable = Literal;
  type Literal = Literal;

  fn new() -> Self {
    let (solver, state) = Solver::init();

    Self {
      current_var: 1,
      solver,
      state: Some(state),
      variables: HashMap::new(),
    }
  }

  fn name() -> &'static str {
    "Kissat"
  }

  fn var(var: Literal) -> Literal {
    var
  }

  fn neg(var: Literal) -> Literal {
    -var
  }

  fn new_var(&mut self) -> Literal {
    let var = self.current_var;
    self.current_var += 1;
    var
  }

  fn add_clause(&mut self, literals: &[Literal]) {
    let mut state = self.state.take().unwrap();
    state = self.solver.add_clause(literals.to_vec(), state);
    self.state.replace(state);
  }

  fn record_variable_name(&mut self, _var: Literal, _name: String) {
    // nothing to be done here
  }

```

```
    fn record_input(&mut self, var: Self::Variable, gate: &GateRef) {
      self.variables.insert(var, gate.clone());
    }
  }

  impl SATSolver for KissatSolver {
    fn new() -> Self {
      Self {}
    }

    fn name() -> &'static str {
      "Kissat"
    }

    fn prepare(&mut self, _gate_model: &GateModel) {
      // nothing to be done here
    }
    fn decide(&mut self, gate_model: &GateModel, gate: &GateRef) -> SATSolution {
      let mut builder = CNFBuilder::<KissatContainer>::new();

      let bad_state_var = builder.visit(gate);
      builder.container_mut().add_clause(&[bad_state_var]);

      for (gate, val) in &gate_model.constraints {
        let constraint_var = builder.visit(&gate.value);
        let constraint_lit = if *val {
          KissatContainer::var(constraint_var)
        } else {
          KissatContainer::neg(constraint_var)
        };
        builder.container_mut().add_clause(&[constraint_lit]);
      }

      let cnf = builder.container_mut();
      let state = cnf.state.take().unwrap();
      let literals = cnf.variables.keys();

      match cnf.solver.solve(state).unwrap() {
        AnyState::SAT(sat_state) => {
          let mut witness = Witness {
            name: String::new(),
            bad_state_gate: gate.clone(),
            gate_assignment: HashMap::new(),
            input_bytes: vec![],
          };
          let mut sat = sat_state;
          for literal in literals.copied() {
            let value = cnf.solver.value(literal, sat).unwrap();
            let assignment = match value.0 {
              Assignment::True => true,
              Assignment::False => false,
              Assignment::Both => false,
            };
            sat = value.1;
```

```
            let gate_ref = cnf.variables.get(&literal).cloned();
            witness
              .gate_assignment
              .insert(HashableGateRef::from(gate_ref.unwrap()), assignment);
          }
          SATSolution::Sat(Some(witness))
        }
        AnyState::UNSAT(..) => SATSolution::Unsat,
        AnyState::INPUT(..) => panic!("expecting 'SAT' or 'UNSAT' here"),
      }
    }
  }
}
```

C Rust Keywords

- *as* - perform primitive casting, disambiguate the specific trait containing an item, or rename items in use statements
- *async* - return a Future instead of blocking the current thread
- *await* - suspend execution until the result of a Future is ready
- *break* - exit a loop immediately
- *const* - define constant items or constant raw pointers
- *continue* - continue to the next loop iteration
- *crate* - in a module path, refers to the crate root
- *dyn* - dynamic dispatch to a trait object
- *else* - fallback for if and if let control flow constructs
- *enum* - define an enumeration
- *extern* - link an external function or variable
- *false* - Boolean false literal
- *fn* - define a function or the function pointer type
- *for* - loop over items from an iterator, implement a trait, or specify a higher-ranked lifetime
- *if* - branch based on the result of a conditional expression
- *impl* - implement inherent or trait functionality
- *in* - part of for loop syntax
- *let* - bind a variable
- *loop* - loop unconditionally
- *match* - match a value to patterns
- *mod* - define a module
- *move* - make a closure take ownership of all its captures

- *mut* - denote mutability in references, raw pointers, or pattern bindings
- *pub* - denote public visibility in struct fields, impl blocks, or modules
- *ref* - bind by reference
- *return* - return from function
- *Self* - a type alias for the type we are defining or implementing
- *self* - method subject or current module
- *static* - global variable or lifetime lasting the entire program execution
- *struct* - define a structure
- *super* - parent module of the current module
- *trait* - define a trait
- *true* - Boolean true literal
- *type* - define a type alias or associated type
- *union* - define a union; is only a keyword when used in a union declaration
- *unsafe* - denote unsafe code, functions, traits, or implementations
- *use* - bring symbols into scope
- *where* - denote clauses that constrain a type
- *while* - loop conditionally based on the result of an expression