Bachelor Thesis

32-bit Support for Bit-Precise Modeling of RISC-U Code

Author:
Patrick Weber

Supervisor:
Prof. Christoph Kirsch

January 2024

Department of Computer Science
Paris Lodron University of Salzburg

Contents

1 Abstract

This thesis presents the 32-bit support for Unicorn, a symbolic execution engine for bit-precise modeling of RISC-V code. The primary motivation for this development was to improve the engine's performance and extend its capabilities to execute 32-bit RISC-U binaries. To this end, we cover the core of the Unicorn engine (on both word and bit level) and discuss the design decisions for implementing the 32-bit support. Our implementation addresses the overflow issues inherent to addition, subtraction and multiplication in 32-bit systems. The results of our experiment evaluation indicate a significant improvement in the engine's efficiency and versatility, demonstrating the potential of this approach in expanding the applicability of the Unicorn engine. We also discuss the challenges we faced during implementation and show ho we resolved them. This work contributes to the ongoing efforts to optimize symbolic execution engines and broadens the scope of their utility in various computing environments. Future work may explore further optimizations and support for other binary types.

2 Introduction

This section motivates our work, summarizes our contributions (Section 2.2), and discusses the collaborative effort with another Bachelor's thesis.

2.1 Motivation

Why advocate for the support of 32-bit architectures when most modern computer systems run on 64-bit architecture? One key reason is performance optimization. Another important consideration is that performing bit-precise reasoning with 64-bit code models is slower compared to semantically equivalent 32-bit code models. This leads us to explore the capabilities of Unicorn, a prominent Symbolic Execution Engine, among other functions. It's crucial to note that advocating for 32-bit support doesn't mean limiting to 32-bit binaries; rather, it involves reconfiguring the underlying model to boost performance. This support includes several crucial features, like full compatibility with 32-bit RISC-U binaries produced by Selfie. Furthermore, it enables running 64-bit binaries on the 32-bit model—a notable concept. Additionally, new functionalities have been added to handle arithmetic overflows, such as addition, subtraction, and multiplication. It's worth noting that the absence of overflow handling for division operations arises from Selfie's built-in support for unsigned division, thus eliminating the possibility of overflow in this scenario.

2.2 Collaboration

This thesis is (partially) a result of a collaboration with a fellow student (Bernhard Haslauer) on the Unicorn project [6]. While we each had distinct responsibilities within the project, we initially undertook the task of comprehending the project and integrating our contributions collaboratively. This shared understanding formed the basis for our joint presentation, where we showcased our collective insights and contributions.

3 Preliminaries

3.1 C* and RISC-U

C* is a tiny subset of the programming language C, which was introduced by the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg in Austria. It is primarily used in the Selfie project [5]. Selfie is a compiler that generates RISC-U binaries, which is also a subset of 14 instructions of the RISC-V architecture.

3.2 Symbolic Execution

Symbolic Execution stands out as a widely embraced technique for thorough program exploration, unveiling the potential to identify inputs that can trigger

program crashes or bugs. This approach facilitates the creation of in-depth unit tests, contributing to the development of more robust and thoroughly tested software. Notably, Symbolic Execution played a crucial role in uncovering a significant portion (1/3) of bugs in Windows 7. The process of Symbolic Execution involves translating the program into a logical formula, ideally in Disjunctive Normal Form (DNF) which looks like that

$$(A \wedge B) \vee (C \wedge D) \vee E$$

or Conjunctive Normal Form (CNF), which we can see below.

$$(A \vee B) \wedge (C \vee D) \wedge E$$

This transformation yields a module, often implemented as an SMT (Satisfiability Modulo Theories) or a similar component, a SMT can be defined like this example.

$$(A \vee B) \wedge (B \rightarrow C) \wedge (C \vee \neg D)$$

This module is designed to pinpoint violations of specific conditions, such as division by zero, illegal memory access, or null pointer references. The efficiency of Symbolic Execution, exemplified by its contribution to bug detection in Windows 7, highlights its value in enhancing software quality and reliability [7] [1].

3.2.1 SMT Solver

An SMT (Satisfiability Modulo Theories) solver serves as a sophisticated software tool meticulously designed to ascertain the satisfiability of logical formulas spanning a variety of theories, including but not limited to arithmetic, arrays, and bit-vectors. This powerful tool is instrumental in tackling intricate problems within the realm of modern computer science, by the means of reasoning effectively over complex scenarios.
Employing decision procedures tailored to specific theories, SMT solvers offer efficient and automated verification of logical constraints within the specified domains. The broad applicability of SMT solvers extends to diverse real-world scenarios, encompassing software and hardware verification, static program analytics, and beyond. Notably, these solvers play a pivotal role in Symbolic Execution, a technique employed within tools like Unicorn.
In the context of Unicorn, SMT solvers contribute to the generation of btor2 files, which can subsequently undergo evaluation by tools like `BTORMC`. This integration highlights the versatility and effectiveness of SMT solvers in addressing various challenges within computer science. Their widespread adoption is a testament to their high efficiency and their indispensable role in advancing the state of the art in this domain. [4]

3.2.2 SAT Solver

A SAT solver is a computer program designed to solve the Boolean satisfiability problem. Given a formula over Boolean variables, e. g. $(x \vee y) \wedge (x \vee \neg y)$, a SAT

solver determines whether this formula is satisfiable, a formula is satisfiable if and only if there are valid values for x and y that make the formula true, or unsatisfiable, otherwise indicating that there are no such values Modern SAT solvers have grown into complex software artifacts involving a large number of heuristics and program optimizations to work efficiently Despite the fact that Boolean satisfiability is an NP-complete problem, efficient and scalable algorithms for SAT were developed during the 2000s. These advancements have contributed to dramatic advances in the ability to automatically solve problem instances involving tens of thousands of variables and millions of constraints.
SAT solvers often begin by converting a formula to conjunctive normal form. They are often based on core algorithms such as the DPLL algorithm, but incorporate a number of extensions and features. Most SAT solvers include time-outs, so they will terminate in reasonable time even if they cannot find a solution. SAT solvers have had a significant impact on fields including software verification, program analysis, constraint solving, artificial intelligence, electronic design automation, and operations research. They are the core component on which satisfiability modulo theories (SMT) solvers are built, which are used for problems such as job scheduling, symbolic execution, program model checking, program verification based on Hoare logic, and other applications. Recently, machine learning approaches have provided a new dimension to solving this challenging problem. For instance, a one-shot model derived from the Transformer architecture has been proposed to solve the MaxSAT problem, which is the optimization version of SAT where the goal is to satisfy the maximum number of clauses. [12]

3.3 Rust

In 2006, Rust originated as a personal side project by Graydon Hoare, who was then employed at Mozilla. Hoare derived the name "Rust" from rust fungi, emphasizing its connection to robustness.
Recognizing the language's potential, Mozilla started sponsoring the project in 2010. The initial pre-alpha version of the Rust compiler was swiftly released in January 2012. Despite its relatively short history, Rust has ascended in popularity among programming languages. In July 2019, it held the 33rd position on the TIOBE Programming Community Index, but by July 2020, it had climbed to the 18th spot. Moreover, according to the Stack Overflow Developer Survey, Rust has consistently been the "most loved" language since 2016.
Positioned as a systems programming language aiming to surpass counterparts like C++, Rust places a primary emphasis on (memory) safety. Over time, it expanded its focus to include optimal performance, adopting the C++ approach of zero-cost abstraction. While excelling in safety and performance compared to other systems languages, Rust boasts versatility with additional advantages. Rust's ecosystem significantly streamlines software projects; a variety of crates (Rust libraries) are available for diverse needs, all easily installable using the official Cargo tool. [2]

4 The Unicorn Engine

In this section, we introduce the Unicorn project, discuss SAT and SMT solvers, and explore the Unicorn tool chain with its two major components Bitlr and Wordlr. Finally, we discuss the the asymptotic complexities of Unicorn.

4.1 Introduction

Unicorn is an open-source tool chain that facilitates symbolic execution of RISC-U machine code through bounded model checking, alongside capabilities for adiabatic and gate-model quantum computing. It converts a C* program P written in a Turing-complete language and converts it into a finite state machine (FSM) over the theory of bit vectors for modeling CPU registers and arrays of bit vectors for modeling main memory. The FSM depicts reachable machine states based on input, where each transition represents the execution of a machine instruction or reading a machine word. A Boolean flag, known as the pc flag, determines which instruction is executed. BEATOR sequentially scans the code twice: first to generate combinational circuits for register and memory updates (data flow) second, to update pc flag (control flow), all without modifying the original code. The BEATOR produces FSMs in the BTOR2 file format, which can be analyzed using the bounded model checker btormc for debugging and validation. QUBOT, unrolls the FSM generated by BEATOR, propagates constants, and transforms it into a satisfiability-preserving QUBO model $\mathcal{B}$. This model ensures that any solution of $\mathcal{B}$ corresponds to input where a program P executes no more than n instructions on machine state M. QUBOT is efficient, operating in $O(n \cdot (n \cdot |P|))$ time and space, with the complexity reducing to $O(n \cdot |P|)$ if P's memory consumption is bounded. QUARC performs similar tasks to QUBOT but generates a semantics-preserving quantum circuit $\mathcal{Q}$, outputting 1 for valid input scenarios and 0 otherwise, represented in the OpenQASM format. Unicorn presents advancements in software translation for quantum computing, compactness of models, and potential speedup in symbolic execution through a novel control and data flow encoding. It offers improvements over existing approaches by incorporating novel techniques and a uniform encoding methodology. [6]

4.2 Motivation

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

  a = 40;
  x = malloc(8);

  *x = 0;
```

```
  read(0, x, 1);

  *x = *x - 47;

  if (*x == 2)
    a = a + *x;
  else
    a = a + (*x * 0);

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

Figure 1: C code of possible division by Zero [6]

4.2 shows our example of what Unicorn can accomplish and where its motivation lies. Here, we observe one possibility where the return exit code is not 0. For the input value 1, we obtain 1 as the return value. Considering that the behavior of this code depends on two variables a and x, there are multiple viable execution paths:

- if the input is 48 then we got a division by zero
- if the input is 50 then we got a division by zero
- if the input is 49 then we got a no zero exit code

Consequently, we can utilize the Symbolic Execution Engine Unicorn to explore various paths and find a way to satisfy a non-zero exit code.

4.3 The Unicorn Toolchain

The Unicorn toolchain consists of various elements, all essential to describe the Finite State Machine. In the following, we describe each comp. in detail.

4.3.1 Wordlr

Wordlr takes x as input and provides y as output. Therefore, Wordlr can be used for bounded execution and any machine state s In the presence of only concrete values, this is straightforward, as all elements of s are simply constants. However, in the presence of symbolic values, some elements of s can become more complex sub graphs. These graphs are closely related to SMT (Satisfiability Modulo Theories) formulae, which we will utilize heavily. For instance, the diamond-shaped graph 3 $\texttt{add}(x, \texttt{mul}(2, x))$ represents the operation $x + 2 \times x$ on a symbolic value x. The nodes `add()` and `mul()` correspond to operations

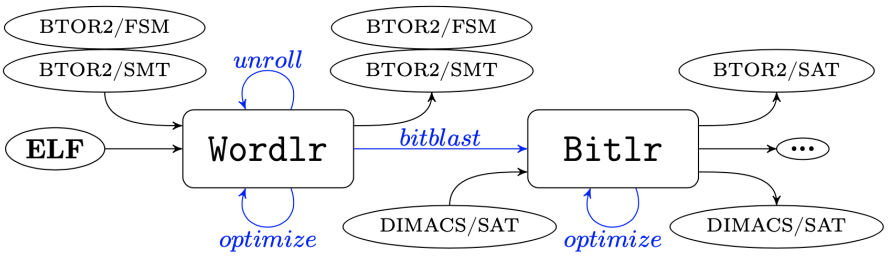

Figure 2: Unicorn's Toolchain [13]

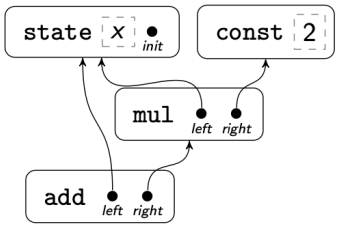

Figure 3: Graph for $\texttt{add}(x, \texttt{mul}(2, x))$ [13]

on fixed-size bit vectors. Any RISC-V register or memory location in s can be described by such a graph
More specially, using the nomenclature of SMT-LIB, there exists a very close correspondence between our WordIR graphs and SMT-LIB formulae using the logic QF_ABV (the Quantifier-Free fragment over the theory of arrays and fixed-size bit vectors). This correspondence enables us to reason over state values using off-the-shelf SMT solvers.
Crucially, we also represent the program p itself as a WordIR graph in such a way that it encodes the exact bit-precise semantics of executing a single instruction in p. The formula encoded by this graph consumes all symbolic values in a state s as arguments and produces the full set of symbolic values for a new state s' as results. We denote this multi-valued formula f_p, and it is important to note that $s' = f_p(s)$ encodes bounded execution of $E_p(s, 1,)$ on any input. This is feasible because f_p is conceptually still a pure data-flow graph; all control flow in p can be represented as a combinational circuit on the program counter encoded in s. We refer to this process as *unrolling* and define it as follows. To improve the overall performance, WordIR employs several methods. As depicted in Figure 2, these methods are called *unroll* and *optimize*.

Unrolling To enhance performance, WordIR uses various methods, including *unroll*. This involves applying the WordIR graph f_p, representing the semantics

of symbolically executing a single instruction in p, to the WordIR graph s, producing s' (with as $s' = f_p(s)$). Unrolling can be applied iteratively, encoding bounded execution of $E_p(s, 3,)$ on any input. The original f_p is stored in memory as a graph fragment, creating a copy before each unrolling step. The resulting graph s_n describes bounded execution of $E_p(s, n,)$, with size $O(n \cdot |f_p|)$. During unrolling, nodes representing program input or safety conditions in f_p are duplicated. Input values and safety conditions are treated separately for each machine cycle. These conditions can be addressed immediately or collected as delayed proof obligations. The set H collects each unrolling steps and grows linearly. After n steps H contains $O(n$ proof obligations. The combination of WordIR graph fragments s and f_p is unbounded, describing the transition function of an FSM over all machine states. Only the state s_n resulting from iterative unrolling is bounded by n unrolling steps.

Optimize The WordIR graph grows over time based on input and instructions, necessitating an overall optimization strategy. Unicorn employs two methods for this: constant folding and whole-program reasoning via SMT solvers. The process of constant folding is an optimization technique applied during compilation that simplifies expressions with unchanging values. This technique ultimately decreases the requirement for runtime computations, leading to a boost in performance. By analyzing and solving fixed expressions beforehand, this approach enhances compiled code efficiency. [10]. For enhanced whole-program reasoning, we exploit the close correspondence between WordIR and SMT. Unicorn maps each node in s_n to SMT-LIB operators from QF_ABV, utilizing Boolector [9] and Z3 [8] solvers. The approach is complete, answering queries efficiently with a timeout option. The size of s_n remains $O(n \cdot |f_p|)$ in the presence of symbolic inputs. [13]

4.3.2 BitLr (Bitblasting)

Once the unrolled and discredited WordIR graph s_n is constructed, it undergoes a transformation into a BitIR graph at the bit level. This conversion, commonly referred to as *bit blasting* , entails the explosion of every bit vector into its constituent bits, treating them as individual Boolean variables. Operations in WordIR are subsequently represented as networks of gates in BitIR, where each gate consumes sets of Boolean values and produces a set of Boolean values.

BitIR comprises 12 distinct types of gates, each generating a single-bit output and consuming up to three bits of input. The transformation from WordIR to BitIR is implemented through a graph traversal in topological order. The resulting composite gate network is denoted as the BitIR graph b_n, and a mapping back to s_n is maintained to correlate reasoning results back to the word level. The bit-blasting process signifies the transition to BitIR, where the size b_n is contingent on the word size w. Typically, $w = 64$ due to 64-bit architecture, but in this implementation, We configured Unicorn to function with $w = 32$,

	Format	Time	Space
WordIR graph for p (unbounded)	BTOR2/FSM	$O(\|p\|)$	$O(\|p\|)$
WordIR as above, w/o array logic	BTOR2/FSM	$O(m \cdot \|p\|)$	$O(m \cdot \|p\|)$
WordIR graph unrolled to given n	BTOR2/SMT	$O(n \cdot \|p\|)$	$O(n \cdot \|p\|)$
WordIR as above, w/o array logic	BTOR2/SMT	$O(n \cdot m \cdot \|p\|)$	$O(n \cdot m \cdot \|p\|)$
BitIR graph converted to CNF	DIMACS/SAT	$O(n \cdot m \cdot \|p\| \cdot w^2)$	$O(n \cdot m \cdot \|p\| \cdot w^2)$

Table 1: Unicorns asymp. compl. [13]

accommodating both 32-bit binaries and 64-bit values down scaled to that size after thorough testing for potential overflows.
While some transitions are linear, others exhibit quadratic behavior with respect to w. For instance, the addition instruction has a linear transition, whereas the multiplication instruction is quadratic. Notably, only the `div`, `mul`, and `rem` instructions demonstrate quadratic behavior, while the others remain linear. The overall size of BitIR can be estimated as $O(|s_n| \cdot w^2)$ with $w = 64$ or 32. This outcome manifests as a logical formula or gate network expressible as a propositional satisfiability problem (SAT), although not necessarily in the format compatible with most SAT solvers.
To address this, Unicorn constructs a CNF for every BitIR given or any subnetwork. This is achieved through a Tseytin transformation applied to the entire BitIR Network [14]. For each gate g representing the operation $A \diamond B$, a new variable (X_g) is introduced. Satisfying the condition ($X_g = A \diamond B$) necessitates the presence of a set of clauses. This process ensures that the clauses are satisfiable if and only if the equation is true. All references to the original gates are replaced with their corresponding variables (X_g) within all relevant clauses, following BitIR language rules.
In summary, Unicorn incurs equivalent time complexities whether constructing through CNF or creating bit-graph networks, as inserting each node into the set requires generating a at most one variable, along with up to eight clusters (each containing a maximum of four literals). Furthermore, such conversion scales proportionally in terms of input size n. [13]

4.4 Asymptotic Complexities

In Table 1, we present an overview of Unicorn's asymptotic complexity. Here, p denotes the input program, m corresponds to memory, n signifies the bound, and w represents the input word size, which can be either 32 or 64 in our case. This table serves as a valuable reference to showcase the notable enhancements achieved with the 32-bit model.
Unicorn's adaptability to different word sizes, especially the improvement observed with the 32-bit model, emphasizes its versatility and efficiency across diverse computing architectures. The parameters p, m, n, and w collectively illustrate Unicorn's bread applicability and its ability to cater to various program specifications and memory configurations. These insights provide valuable information for users seeking optimal performance based on their specific computing

environments and requirements.

5 Implementation

5.1 First tries

My project involved implementing proper 32-bit support in Unicorn, as detailed in the previous sections. To achieve this, substantial modifications were necessary in the model part of the code. While major changes were made to the model, there was no need to alter the overall architecture.
First we attempted a somewhat experimental approach to quickly integrate 32-bit support into Unicorn. We modified two files in the Unicorn codebase, with a focus on the `builder.rs` file. This file plays a crucial role in Unicorn as it builds the WorldlR graph, a concept closely described in the Unicorn documentation and Section 3.3.
In the `builder.rs` file, the primary task was to hack the constants. However, we encountered some challenges, and one particularly time-consuming bug surfaced during this process.
After correctly incorporating all the constants into the code, we maintained a clear record of all the modifications using an unaltered version of Unicorn. This enabled us to keep track of each change made during the implementation process. Subsequently, we navigated to the `mod.rs` file, which writes all information into a `BTOR` file. This `BTOR` file can be executed with tools like `BTORMC` for further analysis and verification.

Listing 1: Execution of the test.btor file with `BTORMC`

```
1 btormc test.btor --kmax 100
```

Listing 1 shows how to execute a `test.btor` file using btormc a max depth of 100 iterations. We observe the execution of the `test.btor` file with a maximum depth of 100 iterations. Listing 1 shows how to execute a `test.btor` file using btormc and a max depth of 100 iterations. Listing 2 shows the output for a 64 bit file and Listing 3 shows the output for 32 bit. Which do basically the same, here we see a big discrepancy in terms of output. Listing **??** and Listing **??** shows the btormc call for a 64-bit and 32-bit text file, which do the same and the corresponding `BTORMC` output

Listing 2: Output of a 64 bit btor file.

```
1 btormc test64.btor --kmax 100
2 0@000000000 1-byte-input@0
  0000000000000000 2-byte-input@0
  000000000000000000000000 3-byte-input@0
  00000000000000000000000000000000 4-byte-input@0
  0000000000000000000000000000000000000000 5-byte-input@0
  000000000000000000000000000000000000000000000000 6-byte-input@0
  00000000000000000000000000000000000000000000000000000000 7-byte-input@0
  0000000000000000000000000000000000000000000000000000000000000000 8-byte-input@0
```

Listing 3: Output of a 32 bit btor file.

```
1 btormc test32.btor --kmax 100
2 0@000000000 1-byte-input@0
  0000000000000000 2-byte-input@0
  00000000000000000000000 3-byte-input@0
  00000000000000000000000000000000 4-byte-input@0
```

Obviously, the input size has shrunk from 8 to 4 bytes, signifying a significant change from 64-bit to 32-bit representation. To achieve this, in the initial attempt with Unicorn, I modified the line responsible for writing all 64-bit bit vectors to the file from

```
writeln!(out, "2 sort bitvec 64 ; 64-bit machine word")?;
```

to

```
writeln!(out, "2 sort bitvec 32 ; 32-bit machine word")?;
```

We transitioned the model from 64bit to 32bit. However, recognizing that this is not a sustainable long-term solution, the subsequent sections will delve into the implementation details of how we integrate these changes into Unicorn to ensure compatibility. Additionally, we introduce some new features.

5.2 Challenges

During my implementation, I encountered several challenges that required careful consideration and resolution. The initial major challenge was comprehending the entire project in its intricacies. Extensive research into Symbolic Execution and Unicorn was necessary to gain a deep understanding. After this crucial initial phase, I embarked on the first attempt at implementation, which, while not overly challenging, laid the foundation for the significant steps taken in my thesis—a substantial learning curve with Unicorn itself.
Navigating through the complexities of Symbolic Execution and Unicorn posed a learning challenge that demanded a deep dive into the project's inner workings. As I progressed, I faced subsequent challenges related to specific modifications, such as ensuring proper 32-bit support and implementing overflow checks. Each challenge provided an opportunity for learning and refinement, contributing to the overall success of the implementation.
At the beginning of my implementation, I faced the first challenge unexpectedly quickly. My initial attempt was to hack in the 32-bit support by manually modifying the model in all necessary files. While this approach worked to a certain extent, I consistently encountered the bad state 'memory-access-between-data-and-heap' at the end of the modification process, which proved difficult to resolve. It took me hours to trace this bug until I realized that I needed to adjust some boundaries in the builder.rs file. This correction successfully resolved the issue, allowing me to proceed with the actual implementation of the code.

After discussing the project with my supervisor, Prof. Kirsch, I realized a problem with the 32-bit overflow check. Initially, I opted for a simple implementation to enhance readability and ensure good performance. However, I didn't thoroughly consider the entire project and the impact of this decision. For instance, if we consider the approach with the 32-bit model, we cannot trigger the multiplication overflow. If you add the signed maximum integer to the same number, you would need 64 bits of space for the result. This number is not representable in the 32-bit model. Consequently, I had to find another solution to detect overflow, which I described in more detail in the implementation section.

5.3 Design Decisions

After the initial attempt to modify the code, we had to reassess our implementation. In this scenario, I had to choose from three different approaches: inline ifs, generics, and building a struct for all variables. Additionally, adjustments were necessary for the RISC-U library from GitHub, which is utilized by Unicorn to decode RISC binaries. [3]

5.3.1 RISC-U Library

Before modifying Unicorn, we had to adopt the RISC-U library. The most important adaption is shown in Listing 4

Listing 4: Most important of RISC-U library

```
if elf.is_lib || !elf.is_64 || !elf.little_endian {
    return Err(RiscuError::InvalidRiscu(
      "has to be an executable, 64bit, static, little endian binary",
    ));
}
```

For 32-bit RISC-U binaries, the error suggested to provide a 64-bit little endian binary. To address this, we initially removed the check for the 64-bit format in the ELF header. However, this alone was not sufficient. Since every instruction supported by Unicorn is 32-bit, there was no need to modify the code for it to function properly.
The only remaining concern was the data segment in the binary. Everything else could be considered to have no impact on the output from this library. Upon examining how the data segment is decoded, we observe that it is simply an array of 8-bit chunks. Handling this is addressed later.
After decoding the program, we receive a struct that looks as depicted in Listing 5.

Listing 5: Example struct in Rust

```
pub struct Program {
    pub code: ProgramSegment<u8>,
    pub data: ProgramSegment<u8>,
    pub instruction_range: Range<u64>,
}
```

Thus, there is no necessity to change. However if a new variable indicating whether it is a 32-bit or 64-bit binary, Unicorn is able to automatically detect which binary format is used. This eliminates the need for any additional input flags to specify whether to run it with 32 or 64 bits. We added the following to the Program struct:

```
pub is64: bool,
```

5.3.2 Inline ifs

Our first approach implements the 32-bit support using so-called inline ifs, as exemplified in Listing **??**.

Listing 6: Inline ifs

```
let data_start = data_section_start &
!(if is_64bit { size_of::<u64>() } else {size_of::<u32>()} as u64 - 1);
```

On the one hand, this approach makes the code readable, requiring only a few adjustments. On the other hand, it can become excessively repetitive, potentially leading to readability issues and bug tracing challenges. Additionally, the performance implications cannot be ignored, as each additional "if" introduces an extra instruction. In order to avoid repetitive code, we chose different approach and explore a new method.

5.3.3 Generics

As an alternative approach, we try to dive a bit deeper into so called generics. Generics in Rust are a powerful feature that prevent code duplication by allowing for abstract stand-ins for concrete types or other properties. This enables the expression of behavior or relationships between generics without knowing what will be in their place when compiling and running the code.
For instance, functions can take parameters of some generic type, instead of a concrete type like `i32` or `String`. This allows the same code to run on multiple concrete values. Listing 7 shows an example of a generic function in Rust.

Listing 7: Generics in Rust

```
fn print<T>(x: T) {
    println!("{:?}", x);
}
```

In the example of Listening, `T` is a stand-in for any type, and the function `print` can take any type as an argument.
Generics are also used in struct and enum definitions, and can be combined with traits to define behavior in a generic way. In listing 8 we see an example of a generic function in Rust.

Listing 8: Generic struct in Rust

```
struct Point<T> {
    x: T,
    y: T,
}
```

The struct, `Point` can be used with any type `T`. Finally, Rust employs a feature called lifetimes—a form of generics that provides the compiler with information about how references relate to each other. Lifetimes ensure the validity of all references in Rust. Listing 7 and 8 offer a glimpse into the power and flexibility of generics in Rust [11]. In the end, however generics are not suitable for our problem. Generics would be more appropriate if dealing with more than two types, hence we decided against generics, and introduce our final choice next.

5.3.4 Structs

Structs in Rust are user-defined data types that allow you to package and name multiple related values that make up a meaningful group. They contain fields that are used to define their particular instance. For example, a struct that stores information about a user account might include fields for active status, username, email, and sign-in count.
To use a struct after it is defined, you create an instance of that struct by specifying concrete values for each of the fields. An example of a struct in Rust is shown in Listing **??**.

Listing 9: Struct in Rust

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

In this example, `User` is a struct that has four fields: `username`, `email`, `sign_in_coun`, and `active`. Each field has a specific type.
This example provide a glimpse into the power and flexibility of structs in Rust. For more detailed information, we refer to the official Rust documentation.
In the end, we concluded that structs are the best way to implement the 32-bit model. For the 32-bit support, we need a few constants and also some if branches. We suggest structs as the most suitable approach because it makes the code more readable and it did not use that much repetitive code.

5.4 Implementation

In this section, we will provide a detailed description of the implementation of the three main components of the 32-bit extension. These components en-

compass the introduction of 32-bit support into the model and the handling of the addition, subtraction and multiplication overflow scenarios. By delving into each aspect, we gain a comprehensive understanding of the modifications made to the codebase to accommodate these features.

5.4.1 32 bit Support

In order to model 32-bit RISC-U binaries with Unicorn, we first had to adopt the model, specifically its nodes. Before doing that, we needed to ensure that every program is parsed correctly. To this end, we edit the function responsible for reading the 8-bit chunks from the decode library.
We modifiy to the for loop over the so-called *dump buffer*.

Listing 10: Dump Buffer 64-bit model

```
dump_buffer
  .chunks(size_of::<u64>())
  .map(LittleEndian::read_u64)
  .zip((data_start..data_end).step_by(size_of::<u64>()))
  .for_each(|(val, adr)| write_value_to_memory(self, val, adr));
```

Listings 10 shows the original code for the *dump buffer* and listing contracts the adoptions. Here, I made changes to all things related to 32-bit. I modified it to the corresponding code snippet.

Listing 11: Dump Buffer 32-bit model

```
dump_buffer
  .chunks(size_of::<u32>())
  .map(LittleEndian::read_u32)
  .zip((data_start..data_end).step_by(size_of::<u32>()))
  .for_each(|(val, adr)| write_value_to_memory(self, val as u64, adr));
```

To ensure compatibility, we perform a cast, as seen in the last line. Additionally, we change the chunk size to 32 bits. Consequently, we now receive an array with all the data inserted, each element being of size 32 bits. Unicorn utilizes this array to write into its memory.
Following this, we adopted certain boundaries and constants. For this purpose, we introduce a new struct that shows all relevant values. The struct is depicted int Listing 12.

Listing 12: Struct 32-bit model

```
struct ModelValues {
    is_64bit: bool,
    run_32bit: bool,
    word_size_mask: u64,
    bits_per_byte: u64,
    size_of: usize,
    bits: u64,
}
```

Each field in the struct has a dedicated responsibility.

- `is_64bit`: Boolean value indicating whether it is a 64 or 32-bit model.
- `run_32bit`: Boolean ensuring that a 64-bit binary will use the 32-bit model of Unicorn.
- `word_size_mask`: Integer indicating the number of bytes in a word; for 64-bit, it is 8.
- `bits_per_byte`: Integer specifying the number of bits in a byte, always set to 8.
- `size_of`: Used to chunk down an array into a certain size, either 64 or 32 bits.
- `bits`: Indicating how many bits the model has, either 64 or 32.

This struct is generated at the top of the model generation and is utilized throughout the entire model. It is also used to initialize all boundaries, which are employed for detecting bad states related to memory access errors. Additionally, we change stack initialization. Therefore, a new function `prepare_unix_stack32bit` handles the overall preparation of the stack in the Unicorn model. In this function, the arguments of the program are casted to 32-bit values to ensure compatibility with the model. Furthermore, all pointers to the values on the stack are prepared to facilitate access.
I also had to truncate the input nodes for the model. The 32-bit model has only 4 nodes ranging from 1 to 4 bytes, whereas the 64-bit model has nodes ranging from 1 to 8 bytes. In this case, I used a simple if statement to exclude nodes if the model is 32-bit. Towards the end, we also modified the `mod.rs` file, which is responsible for writing the model to a `BTOR` file. In this context, adjustments were required for the word node declaration, specifying whether a word has 32 or 64 bits. Listing 13 and Listing 14 shows the original and the adapted code, respectively

Listing 13:

```
writeln!(out, "2 sort bitvec 64 ; 64-bit machine word")?;
writeln!(out, "3 sort array 2 2 ; 64-bit virtual memory")?;
```

Listing 14:

```
writeln!(out, "2 sort bitvec 32 ; 32-bit machine word")?;
writeln!(out, "3 sort array 2 2 ; 32-bit virtual memory")?;
```

Consequently the header of the `BTOR` file is configured correctly. Finally we updated some method signatures, which requires careful attention in other parts of the Unicorn source code. These changes necessitate a thorough review and adaptation in various sections to ensure seamless integration and functionality.

5.4.2 Addition and Subtraction Overflow

For the detection of an overflow, we have to generate a so-called *bad state*, which indicates a violation of certain constraints. Examples for bad states include illegal memory access, division by zero, or a non-zero exit code, indicating a program crash. To achieve this, we need to check if an addition exceeds the maximum size of a 32-bit integer. The maximum signed 32-bit integer is $2^{31} - 1 = 2147483647$. Our first idea was to utilize existing infrastructures in the form an `ugt` (unsigned greater than) condition to check if the result of an integer addition or subtraction surpasses this value.
To capture a bad state in a specific condition, we first need to create a flow. Therefore we introduced a new flow named `addition_overflow_flow`. This is a node in the Unicorn model that stores the program counter (`pc`) flag and the result of the addition. This node traverses the entire model, capturing all additions, and eventually, we generate a bad state with it. The code for the bad state is shown in Listing 15

Listing 15: Bad State Addition

```
let check_addition_overflow =
 self.new_ugt(self.addition_overflow.clone(),
    self.max32bitInteger.clone());
 self.new_bad(check_addition_overflow,
    "addition-subtraction-overflow");
```

Here we see the comparison with the `addition_overflow`. So if, in any case, the constraint is violated throughout the program, this bad state gets triggered, and Unicorn repaints an error accordingly.

Listing 16: Bad State Addition Output

```
    Bad state 'addition-subtraction-overflow[n=79]' is satisfiable!
```

Although this approach works at first sight, we realized that it only captures overflows within the 64-bit model, in the 32 bit model the overflow is undetected. In this model we would have an overflow. Hence, 23 contemplated a different approach. How can we detect an overflow within an addition or subtraction? The easiest way is to look at the MSB (Most Significant Bit). The MSB indicates the sign of an integer. If this bit is 1, it is a negative number, and if it is 0, it is positive.
A overflow can be detected if the MSB of both operands is the same, but the MSB of the result is not the same. To identify this, we implemented the code shown in Listing 17 for each instruction.
This approach ensures that the most significant bit alignment is consistent between operands and the result, allowing for the detection of overflow conditions. This check is vital for maintaining the integrity of arithmetic operations within the program.

Listing 17: Overflow check for addition , subtraction accordingly

```
fn check_addition_overflow(&mut self, rs1: NodeRef,
rs2: NodeRef, rd: NodeRef) {
 let bits = self.new_const(self.model_values.bits - 1);
 let msb_rs1 = self.new_srl(rs1, bits.clone());
 let msb_rs2 = self.new_srl(rs2, bits.clone());
 let msb_rd = self.new_srl(rd, bits);

 let equal_msb = self.new_eq(msb_rs1.clone(), msb_rs2);
 let overflow = self.new_neq(msb_rs1, msb_rd);

 let overflow_occurred = self.new_and_bit(equal_msb, overflow);

 self.addition_overflow = self.new_ite(
   self.pc_flag(),
   overflow_occurred,
   self.addition_overflow.clone(),
   NodeType::Bit,
   );
}
```

First we perform a left shift on all operands and the result by the number of bits minus 1 to obtain the most significant bit. Subsequently, we check if the MSB of the operands is the same, followed by a comparison with the MSB of the result. Finally, we concatenate all the bits to form a bad state, which we verify to be non-zero within a bad state. This overflow check is designed to be applicable for both 32 and 64-bit scenarios.
This method ensures a rigorous examination of the most significant bit alignment, establishing a robust mechanism for detecting overflow conditions in arithmetic operations across different bit architectures.

5.4.3 Multiplication Overflow

To check for an overflow during a multiplication is a bit more tricky. We cannot simply check if exceeds the max 32-bit signed integer. Hence, we had to find a different approach, which is shown in Listing 18:

Listing 18: Overflow check for addition , subtraction accordingly

```
fn check_multiplication_overflow(&mut self, rs1: NodeRef, rs2:
NodeRef, rd: NodeRef) {
    let bits = self.new_const(self.model_values.bits - 1);
    let msb_rs1 = self.new_srl(rs1, bits.clone());
    let msb_rs2 = self.new_srl(rs2, bits.clone());
    let msb_rd = self.new_srl(rd, bits);

    let msb_eq = self.new_eq(msb_rs1, msb_rs2);
    let overflow_minus = self.new_neq(msb_rd, self.zero_word.clone());

    let overflow_occured_msb = self.new_eq(msb_eq.clone(),
    overflow_minus.clone());
    let overflow_occured_not_msb = self.new_eq(msb_eq, overflow_minus);

    let overflow_occurred = self.new_or(overflow_occured_msb,
    overflow_occured_not_msb);

    self.multiplication = self.new_ite(
        self.pc_flag(),
        overflow_occurred,
        self.multiplication.clone(),
        NodeType::Bit,
        );
}
```

In this approach, we extract the most significant bit (MSB) from the operands and the sum during addition, allowing for efficient bit-wise comparison. The comparison is crucial for detecting overflow conditions, as illustrated in Table 1, which presents all possible combinations of signs for multiplication The MSB of

MSB RS 1	**MSB RS 2**	**MSB RD**
1 (Negative)	1 (Negative)	0 (Positive)
1 (Negative)	0 (Positive)	1 (Negative)
0 (Positive)	1 (Negative)	1 (Negative)
0 (Positive)	0 (Positive)	0 (Positive)

Table 2: Combination of signs for multiplication and overflow detection

the result (**MSB RD**) is determined by the signs of the operands (**MSB RS 1** and **MSB RS 2**). This information is crucial for overflow detection. If both MSBs of the operands are the same, the result must be positive, so we check if it is zero. If the MSBs are different, the result must be negative, and we check if it is one.
This bit-wise comparison strategy enhances performance and ensures proper handling of overflow scenarios during multiplication.

5.5 Experimental Evaluation

After completing the implementation, a thorough testing and evaluation phase is essential. The tests can be divided into three categories, corresponding to different aspects of the implementation, support for 32-bit operations and han-

dling detection of positive and negative overflows.

5.5.1 32-bit Support

Initially, a crucial step in the verification process involved rigorous testing to ensure the accuracy and reliability of the newly implemented 32-bit support. To conduct this examination, a meticulous approach was adopted, focusing on the comparison of outputs generated by object files when processed through both the original Unicorn implementation and Unicorn with 32-bit support. The chosen file for this intricate assessment was meticulously selected for its suitability in capturing the nuances of 32-bit support.
This comprehensive testing phase served as a foundational measure to affirm the integrity and compatibility of the modifications made to Unicorn, providing confidence in its ability to handle 32-bit scenarios effectively. For this testing, I utilized the `division_by_zero.c` file available in the Unicorn repository. The structure of the file is shown in Listing 19:

Listing 19: Division by Zero. [6]

```
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;
}
```

We took the file and generated two object files using the selfie compiler [5]. The compilation process resulted in two distinct files: `div64.o` and `div32.o`. Subsequently, when processing `div64.o` with the original version of Unicorn, yields the output shown in Listing 19.

Listing 20: Division by zero output from Unicorn.

```
Bad state 'division-by-zero[n=82]' is satisfiable!
Bad state 'addition-subtraction-overflow[n=76]' is satisfiable!
Bad state 'invalid-syscall-id' is satisfiable!
Bad state 'memory-access-below-data' is satisfiable!
Bad state 'memory-access-between-data-and-heap' is satisfiable!
Bad state 'memory-access-between-max-and-dyn-heap' is satisfiable!
Bad state 'memory-access-between-heap-and-stack' is satisfiable!
Bad state 'memory-access-between-dyn-and-max-stack' is satisfiable!
Bad state 'memory-access-above-stack' is satisfiable!
Bad state 'division-by-zero' is satisfiable!
Bad state 'remainder-by-zero' is satisfiable!
Bad state 'non-zero-exit-code' is satisfiable!
```

For Unicorn with 32-bit support, we obtained exactly the same results with either 32 or 64-bit inputs. Consequently this test was successful. However, to ensure the robustness of the implementation, we conducted this test with multiple files in the Unicorn repository.
This is one part of the test. The next step involved creating the `BTOR` file for both the 32-bit model and the original version of Unicorn. Subsequently, we utilized `BTORMC` for execution and examined the output. The most important lines are similar to Listing 21

Listing 21: Division by zero output from `BTORMC`.

```
sat
b12
010 00110010 1-byte-input[n=68]0
@0
```

Here, we observe the witness that encounters the bad state and identifies the triggered bad state, in this instance labeled as `b12`. Upon comparison with the original version, the results was identical. This rigorous testing and evaluation process were systematically conducted with multiple files, affirming the robustness and effectiveness of the 32-bit support.

5.5.2 Overflows

After implementing the overflow support, we experimentally evaluated it using the methodology of 32-bit tests. To provoke overflow scenarios in Unicorn, I created multiple files. An example of such a file is illustrated in Listing 22:

Listing 22: C code for an overflow

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

    a = 2147483647;

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

    read(0, x, 1);
    a = a + *x;

    return 0;
}
```

Ib Listing 22 we observe that a is nearly the maximum signed 32-bit integer. I ran Unicorn with this file for both 32-bit and 64-bit configurations. We expect to see the bad state with 32-bit but not with 64-bit. The output for the 32-bit version is seen in Listing 23:

Listing 23: Overflow output from Unicorn.

```
Bad state 'addition-subtraction-overflow[n=79]' is satisfiable!
Bad state 'invalid-syscall-id' is satisfiable!
Bad state 'memory-access-below-data' is satisfiable!
Bad state 'memory-access-between-data-and-heap' is satisfiable!
Bad state 'memory-access-between-max-and-dyn-heap' is satisfiable!
Bad state 'memory-access-between-heap-and-stack' is satisfiable!
Bad state 'memory-access-between-dyn-and-max-stack' is satisfiable!
Bad state 'memory-access-above-stack' is satisfiable!
Bad state 'remainder-by-zero' is satisfiable!
Bad state 'non-zero-exit-code' is satisfiable!
Bad state 'addition-subtraction-overflow' is satisfiable!
```

In the 32-bit output, we can clearly see that the bad state 'addition-subtraction-overflow' was triggered. However, in the 64-bit version, the first line is missing, confirming that the bad state is functioning as intended.

6 Conclusions

At the end of this thesis, it is time to provide a brief overview of all that has been accomplished. I successfully implemented the 32-bit support and adjusted the model accordingly. Fortunately, I did not need to modify the bit-blasting, which was already well-implemented, capable of handling word sizes of any size and transforming them into the correct bit graph. Overall, this enhancement

contributes to improving the performance of Unicorn. It's essential to highlight that Unicorn now supports additional bad states, increasing its usability. Detecting overflow is not always a glaring error in code but can be challenging to discover, making it a significant benefit to the project.

As we conclude this thesis, there are promising avenues for future research and development. The support for 32-bit systems can be further extended, refining Unicorn's capabilities in this domain. Given the growing importance of Symbolic Execution, ongoing research in this area is essential.

Future work could focus on optimizing and enhancing the handling of specific 32-bit instructions, addressing challenges unique to this architecture. Additionally, evaluating Unicorn's performance with real-world 32-bit binaries could identify areas for improvement. Adapting Unicorn to support emerging technologies and seamlessly integrating it into various analysis tools are also potential areas for exploration.

Practical applications, especially in vulnerability analysis and reverse engineering, could be explored to understand the real-world impact of enhanced 32-bit support in Unicorn. In summary, the future holds opportunities for both technical advancements and broader applications, fostering continuous innovation in binary analysis and symbolic execution tools.

7 Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Salzburg, March 20, 2024 .....

8 References

[1] Roberto Baldoni et al. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[2] William Bugden and Ayman Alahmar. "Rust: The programming language for safety and performance". In: *arXiv preprint arXiv:2206.05503* (2022).

[3] CK Systems Group. *RISC-U*. `https://github.com/cksystemsgroup/riscu`. Accessed: March 20, 2024.

[4] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[5] Computational Systems Group. *Selfie*. `https://github.com/cksystemsteaching/selfie`. 2024.

[6] Computational Systems Group. *Unicorn*. `https://github.com/cksystemsgroup/unicorn`. 2024.

[7] Donald Knuth. *Knuth: Computers and Typesetting*. URL: `http://www-cs-faculty.stanford.edu/~uno/abcde.html`. (accessed: 01.09.2016).

[8] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[9] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0". In: *J. Satisf. Boolean Model. Comput.* 9.1 (2014), pp. 53–58. DOI: `10.3233/sat190101`. URL: `https://doi.org/10.3233/sat190101`.

[10] Seminar Effiziente Programmierung and Marcel Papenfuss. "Compiler-Optimierungen". In: (2019).

[11] The Rust Project. *The Rust Programming Language - Generics*. Accessed: March 20, 2024. Year. URL: `https://doc.rust-lang.org/book/ch10-01-syntax.html`.

[12] Feng Shi et al. "Transformer-based machine learning for fast SAT solvers and logic synthesis". In: *arXiv preprint arXiv:2107.07116* (2021).

[13] Michael Starzinger. 2023.

[14] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983), pp. 466–483.