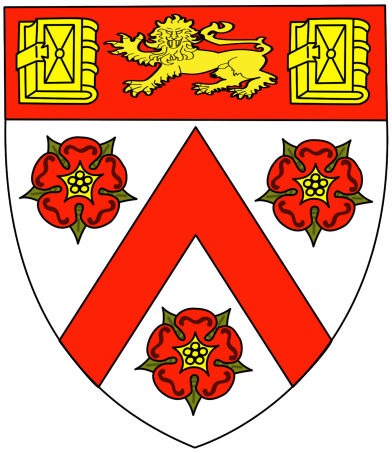
**Matej Urban**

# Formal Verification of ELF Relocations

Computer Science Tripos – Part II



Trinity College

2025

# Declaration of originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 2393G, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Date *May 16, 2025*

# Proforma

**Candidate Number** 2393G

**Project Title** Formal Verification of ELF Relocations

**Examination** Computer Science Tripos – Part II, 2025

**Word Count** 11994[1]

**Code Line Count** 5065[2]

**Project Originator** Prof. Peter Sewell and Dr. Stephen Kell

**Project Supervisor** Prof. Peter Sewell and Dr. Stephen Kell

## Original Aims of the Project

This project investigates the *semantic relocatability* of relocatable ELF files — an often desired property that allows their sections to be placed at different memory addresses without altering runtime behaviour. The project focuses on compiled C code, and leverages debug information.

The core goal is to extend existing tools to enable symbolic execution of such files under varying memory layouts and evaluation of their debug information. As an extension, the project aims to develop an automatic verifier for semantic relocatability. Since automatic verification of arbitrary programs is undecidable, the scope must be restricted to a narrower class of programs.

## Work Completed

The project was a success. The core goals were met, and a working version of the semantic relocatability verification tool was implemented. The core components were successfully tested on a large collection of external test cases. The verification tool was successfully applied to programs using integers and pointers. It lacks support for composite data types such as structs, unions, and arrays, which could be added with some engineering effort. A more fundamental challenge is loops. Supporting them will require adjusting the current approach, which includes some open questions.

## Special Difficulties

None.

[1] Computed using `texcount`.

[2] Computed using `git diff` against the original repositories that were modified. Additional code used for evaluation was counted using `cloc`.

# Contents

# 1 | Introduction

When compiling a C source file, the compiler typically produces a *relocatable file*, which can be combined with other relocatable files to produce an executable. A (static) linker determines how the sections of these files should be arranged in memory. Based on the arrangement, it must perform *relocations* — rewrite the addresses that appear in the program so it functions correctly when executed. We expect most relocatable files to behave consistently at runtime, regardless of the positions of their sections in memory. In this project, I introduce the term *semantic relocatability* to describe this property, and explore methods to formally verify it.

## 1.1 Motivation — semantic relocatability

We consider a program file *semantically relocatable* if its observed behaviour is not affected by the memory addresses where its sections are placed. We expect most programs to have this property, although some may knowingly violate it (e.g., when using hash tables indexed by pointer values). Sometimes, the property is violated unintentionally due to improper pointer manipulation. Figure 1.1 shows an example found in the GCC test suite [5]. Automatic verification of this property for arbitrary programs is (obviously) impossible, but programs rarely perform complicated operations with pointer values, making the verification viable.

The main approach used in this project is symbolic execution. The idea is to represent unknown values (e.g., inputs of the program, or in our case, the sections' memory addresses) by symbols, and instead of performing a computation with concrete values, operate on expressions involving the symbols [6]. If the program contains branching, symbolic execution explores all possible execution paths. This lets us analyse how the memory addresses affect the execution.

The implementation is based on the read-dwarf tool developed by researchers at the University of Cambridge, to explore binary validation assisted by debug information. It provides a framework for symbolically executing ELF files, and uses debug information to extract information such as the type signatures of functions. [8]

The debug information annotates the binary with source-level details that are otherwise lost during compilation, such as the representation of variables and their types. It is stored in a compact DWARF format. Extracting the information requires non-trivial processing, involving the evaluation of programs in special DWARF-specific stack machine languages. [4]

When verifying semantic relocatability, the debug information can be used to generate reasonable constraints (e.g., that pointers may be affected by section addresses, but integers should not) that allow the verification to be fully automatic, requiring no manual annotations.

```c
int ns_name_skip (unsigned char **x,
                  unsigned char *y) {
  *x = 0;
  return 0;
}

unsigned char a[2];

int dn_skipname(unsigned char *ptr,
                unsigned char *eom) {
    unsigned char *saveptr = ptr;
    if (ns_name_skip(&ptr, eom) == -1)
        return (-1);
    return (ptr - saveptr);
}

int main(void) {
  if (dn_skipname (&a[0], &a[1]) == 0)
    abort ();
  exit (0);
}
```

**(a)** Original (reformatted)

```c
unsigned char a;

int dn_skipname(unsigned char *ptr) {
    unsigned char *null = 0;
    return (null - ptr);
}

int main(void) {
    if (dn_skipname(&a) == 0)
        abort ();
    exit (0);
}
```

**(b)** Equivalent simplified version

**Figure 1.1:** GCC C torture test `980701-1.c`, violating semantic relocatability. The problem is best seen on the simplified version. The return type of `dn_skipname` is `int` (assume 32 bits), which causes an overflow for large values of `ptr`. When the address of `a` is any multiple of $2^{32}$, the function call `dn_skipname(&a)` returns zero and the test fails.

## 1.2 Aims

Prior to this project, read-dwarf only supported executable files, and had no concept of relocations. This project extends it to support relocatable files, allowing the following:

- Symbolic execution of relocatable ELF files with sections' addresses represented symbolically.
- Evaluation of DWARF debug information from relocatable ELF files. Parts of the debug information, such as the locations of global variables, depend on the addresses of sections, which we represent symbolically.

As an extension of the project, I build an automated verification tool for *semantic relocatability*, capable of verifying simple programs. The tool combines symbolic execution with bisimulation to verify the property in a semiformal way. It relies on exhaustive path exploration during symbolic execution, limiting it to programs that always terminate. Additionally, the current implementation does not support composite data types, which could be addressed in later versions.

# 2 | Preparation

## 2.1 Overview of ELF

Executable and Linkable Format (ELF) is the standard format for executables and object files on Unix systems. An ELF file contains sections of different kinds.

Some sections consist of data that is loaded into memory for the program's execution (e.g., `.text` contains the program instructions, `.data` contains global variables). Others contain control information needed for linking, or debug information.

Symbols mark addresses within the sections that other parts of the ELF file can refer to, such as functions and global variables. They are described in a symbol table (in a dedicated `.symtab` section).

Programs are typically compiled into *relocatable* ELF files that do not yet describe how sections should be arranged in memory. A static linker takes one or more relocatable files, assigns a memory location to each section, and rewrites references to symbols in a process called relocation. This results in a single *executable* ELF file.

### 2.1.1 Example function

```
long long x = 1;

void f(long long a) {
  x = a;
}
```

**Figure 2.1:** Example C file

The file in Figure 2.1 gets compiled into a relocatable file, placing `x` into `.data` and the code of `f` into `.text`. The addresses of these sections are only determined when a static linker processes the file. Relocations ensure that, no matter where the sections end up, executing the function always results in setting `x` to `a`, by rewriting the machine instructions that access `x` to refer to the memory location where `x` is located at runtime.

This example will be used throughout this chapter and the next one.

### 2.1.2 Relocation

Figure 2.2a shows the disassembly of the function from Figure 2.1, compiled into an AArch64 relocatable file, with the following instructions:

1. `adrp` loads the page address of `x` (aligned to 4KB) into the register `x8`.

2. `str` takes the page address from `x8` and adds the page offset of `x` to obtain the full address. Then it writes the value in register `x0` to that address.

It contains relocation entries (red), that describe how to perform relocations. Both the `adrp` and `str` instructions require the address of `x`, so they have appropriate relocation entries.

```
0:  90000008  adrp  x8, 0
0:  R_AARCH64_ADR_PREL_PG_HI21    x
4:  f9000100  str   x0, [x8]
4:  R_AARCH64_LDST64_ABS_LO12_NC  x
c:  d65f03c0  ret
```

**(a)** Relocatable file

```
4000e8: 90000088  adrp  x8, 410000
4000f0: f907f500  str   x0, [x8, #4072]
4000f4: d65f03c0  ret
```

**(b)** Executable file

**Figure 2.2:** Comparison between objdump of a relocatable and executable file

Each relocation entry describes the relocation *type* (R_AARCH64_ADR_PREL_PG_HI21, R_AARCH64_LDST64_ABS_LO12_NC), defining what kind of action must be performed, a *symbol* (`x`) whose address is used to compute the value to be plugged in, and an optional *addend*, also used in computing the value. Relocation types are machine-specific, defined by the appropriate Application Binary Interface (ABI) specification. Table 2.1 shows a fragment from the specification for AArch64.

The program after performing the relocations is shown in Figure 2.2b. The immediate values that were changed by relocation are highlighted in red.

| **Name** | **Operation** | **Comment** |
|---|---|---|
| R_AARCH64_ADR_PREL_PG_HI21 | Page(S+A) - Page(P) | Set an ADRP immediate value to bits [32:12] of the X; check that $-2^{32} <= X < 2^{32}$. |
| R_AARCH64_LDST64_ABS_LO12_NC | S + A | Set an LD/ST immediate value to bits [11:3] of X. No overflow check. |
| R_AARCH64_ABS64 | S + A | Write bits [63:0] of X at byte-aligned place P. No overflow check. |

S = address of the *symbol*
A = *addend*
P = address of the place being relocated
X = result of the relocation operation
Page(expr) = page address of the expression expr, defined as (expr & ~0xFFF)

**Table 2.1:** Specification of selected AArch64 relocations [3]

### 2.1.3 DWARF

Most compilers have the option of including debug information in the compiled binary. It describes, among other things, which instructions correspond to which lines of the program, and how variables are placed in memory. This is used by debuggers such as `gdb` to set breakpoints and examine variables during execution.

It is encoded using the DWARF format [4] in ELF sections with the prefix `.debug`. Figure 2.3 shows parts of the debug information of the program from Figure 2.1, that can be used to determine the location of variables `x` and `a`. The locations are described in a DWARF-specific stack machine language. Languages such as this one are used throughout DWARF to express the debug information in a compact way. Interpreting the debug information thus requires evaluating programs in these languages. A more complicated location description is shown in Figure 2.4.

```
<0><b>: DW_TAG_compile_unit
   ...
   <12>   DW_AT_name        : example.c
   ...
   <1e>   DW_AT_low_pc      : 0x4000e8
   <26>   DW_AT_high_pc     : 0xc
   <1><2a>: DW_TAG_variable
      <2b>   DW_AT_name        : x
      ...
      <35>   DW_AT_location    : DW_OP_addr 410fe8
   ...
   <1><46>: DW_TAG_subprogram
      <47>   DW_AT_low_pc      : 0x4000e8
      <4f>   DW_AT_high_pc     : 0xc
      ...
      <55>   DW_AT_name        : f
      <2><5b>: DW_TAG_formal_parameter
         <5c>   DW_AT_location    : DW_OP_reg0
         <5e>   DW_AT_name        : a
         ...
```

**Figure 2.3:** DWARF debug information (some attributes omitted). Memory addresses are shown in red. In a relocatable file, these must have associated relocation entries. The locations of variables are described using operations: `DW_OP_addr` - value at some address in memory, `DW_OP_reg0` - value in register 0.

```
DW_AT_location  : DW_OP_breg20 0x0; DW_OP_constu 0x1; DW_OP_and; DW_OP_stack_value
```

**Figure 2.4:** More complicated location description (from pKVM [2]). The value of the variable is obtained by taking the value in register 20 and performing a bitwise and with a literal 0x1.

As seen in Figure 2.3, some parts of DWARF reference memory locations. In a relocatable file, they must have the corresponding relocation entries (Figure 2.5).

```
  Offset             Type         Sym. Name + Addend
000000000001e  R_AARCH64_ABS64   .text + 0           // start of the compilation unit
000000000037   R_AARCH64_ABS64   x + 0               // location of the variable x
000000000047   R_AARCH64_ABS64   .text + 0           // start of the function f
```

**Figure 2.5:** Relocations in DWARF

Besides its usual use case, debug information can be useful for validating binaries with respect to their source implementation. The debug information describes how the binary represents elements from the source program, which would otherwise be difficult to infer.

## 2.2 Symbolic execution

Symbolic execution is a method for abstractly executing a program to determine its behaviour for different inputs. The key idea is to represent all unknown values, such as the inputs, using symbols that stand for arbitrary values. A symbolic executor executes the program as normal, but instead of concrete values, it operates on symbolic expressions over the input symbols. [6]

When a branch is encountered that depends on an unknown symbolic value, the executor proceeds to explore both paths. The executor keeps a set of path conditions that lead to choosing each particular path.

The result of the symbolic execution can be viewed as a state tree, capturing all possible executions of the program. Each node describes a symbolic state, together with its path conditions.

SMT solvers can be used to simplify the state tree. They can simplify symbolic expressions and remove impossible paths by checking the satisfiability of path conditions. This is usually done during execution to improve performance. SMT solvers can be further used to ask questions about the possible outcomes of each execution path.

## 2.3 Simulation

Simulation is a formal method for establishing a correspondence between two systems, often modelled as state transition systems. Simulation shows that one system can mimic the behaviour of another. A bisimulation is a stronger symmetric version, showing that the systems can simulate each other. Formally:

**Definition 2.3.1** *Given two labelled state transition systems, with sets of states $S$ and $S'$ respectively, a relation $R \in S \times S'$ is a simulation if for every pair of states $(p, q) \in R$:*

- *for every transition $p \xrightarrow{\lambda} p'$, there is a transition $q \xrightarrow{\lambda} q'$, such that $(p', q') \in R$.*

*It is a bisimulation if for every pair of states $(p, q) \in R$:*

- *for every transition $p \xrightarrow{\lambda} p'$, there is a transition $q \xrightarrow{\lambda} q'$, such that $(p', q') \in R$, and*
- *for every transition $q \xrightarrow{\lambda} q'$, there is a transition $p \xrightarrow{\lambda} p'$, such that $(p', q') \in R$.*

A bisimulation can be combined with symbolic execution to prove the equivalence between two programs. For simple programs, when the complete (finite) execution tree can be constructed, we can use the following procedure:

1. Run symbolic execution on both programs, obtaining two state trees. Each program can be seen as a state transition system consisting of the initial state and the leaf states of the state tree. It begins in the initial state and can transition to any leaf state as long as the path conditions are satisfied.
2. Construct a relation between the leaf states.
3. Verify the bisimulation by checking that the path conditions of states bound by the relation are equivalent.
4. Check that the outputs and side effects of states bound by the relation are equal.

We will use this approach to verify the equivalence between two instances of the same program, with varying memory layouts. It cannot be used for programs with unbounded loops, since their execution tree is infinite. It is also not suitable for more complex programs whose execution tree is too large to be efficiently constructed. There are methods which do not require constructing the whole execution tree, but they were not used in this project (discussed in §5.2).

## 2.4 Starting point

This project builds on multiple tools. The most important are:

- *linksem* — for reading ELF files, including the DWARF debug information,
- *read-dwarf* — for symbolic execution of binaries and further processing of DWARF on top of linksem,
- *Isla* — a dependency of read-dwarf allowing symbolic execution of machine instructions.

The majority of my work is deeply integrated into linksem and read-dwarf. I was not familiar with the tools before this project, so I had to learn about them before and during implementation.

### 2.4.1 Linksem

Linksem is an executable specification of ELF, including the DWARF debug information [7]. It is used by read-dwarf as a tool for parsing ELF files and interpreting DWARF. It is written in a custom specification language *Lem*, which allows it to be compiled into OCaml as well as different theorem prover backends. For this project, only the OCaml backend is relevant.

There are functions to parse the high-level structure of the ELF files, as well as the contents of standard sections, such as symbol table and relocation sections.

The DWARF module defines a representation of DWARF and functions to parse it from the appropriate ELF sections. It further provides functionality for evaluating the DWARF expressions and a number of analysis functions. These are used to obtain the full expanded debug information.

### 2.4.2 Read Dwarf

Read-dwarf is an experimental tool intended for validating ELF files compiled from C with the help of DWARF [8]. It contains scripts for pretty-printing the debug information parsed by linksem and for symbolically executing ELF binaries. This section will focus on the symbolic execution, since it will be relevant in the implementation section.

#### Symbolic execution

Read-dwarf can symbolically execute functions in an ELF file. It starts by constructing an initial state according to the function's ABI. Then, a single execution step proceeds as follows:

1. Read the PC register from the state.
2. Look up the instruction at the PC address and obtain its **instruction traces**. A trace describes the effects of an instruction on the state.
3. For each trace, apply it to the state, obtaining a new state.

The result of an execution step is one or more new states. Most instructions produce only one new state, but branching instructions produce multiple. The execution then continues recursively from each of the new states. The result of the execution is an execution tree of states.

### Obtaining a trace

Read-dwarf uses Isla [1] to obtain traces for instructions. Isla is a symbolic execution tool that uses formal instruction set architecture (ISA) specifications to determine the effects of machine instructions. The possible effects are described using a set of traces, each consisting of a sequence of commands. Isla provides a script Isla-client, which allows other processes to communicate with it.

The opcode of the current instruction is sent to Isla-client, which sends back a trace in a text format (**Isla trace**). The trace is parsed and simplified. It is further processed into a simplified format, more convenient for the symbolic executor (**instruction trace**). This pipeline is illustrated in Figure 2.6.

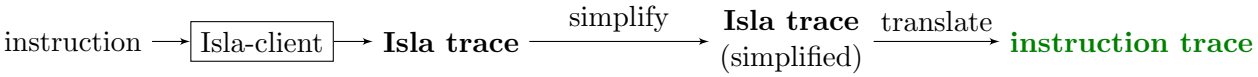



**Figure 2.6:** Instruction trace pipeline

### Three types of traces and expressions

Read-dwarf's symbolic execution involves three different kinds of traces and symbolic expressions. Two of them are shown in Figure 2.6 and the third - **memory trace** - is used in the state representation of memory. I will explain the purpose of each and outline the differences between them.

Each trace is a sequence of commands that symbolically describe a series of read/write operations. They use symbolic expressions (using SMT constraints) to express the values used by each operation. The form of symbolic expressions also differs between the three.

```
(trace
  (declare-const v1 (_ BitVec 64))
  (declare-const v2 (_ BitVec 64))
  (read-reg |R8| v1)
  (read-reg |R0| v2)
  (define-const v3 ((_ extract 51 0 ) ( bvadd v1 #x0000000000000fe8)))
  (define-const v4 ((_ zero_extend 12) v3))
  (write-mem v4 v1 4)
)
```

**(a)** Isla trace (simplified)

$$\mathsf{WriteMem}_8(\mathrm{Register}(\mathrm{R8})[0:51]{+}0\mathrm{xfe8}{:}52,\quad \mathrm{Register}(\mathrm{R0}))$$

**(b)** Instruction trace

**Figure 2.7:** Traces for instruction `str x0, [x8, #4072]`

$$\mathsf{Write}_8(0\mathrm{x410fe8}, \mathrm{Arg}(0))$$

**Figure 2.8:** Memory trace after executing the function from Figure 2.2b

**Notation** To concisely express the symbolic expressions, the following notation is used (also used in read-dwarf's outputs).

| | | |
|---|---|---|
| $e[lo:hi]$ | = | extract bits $lo$ to $hi$ inclusive from $e$ |
| $e_1.e_2$ | = | concatenate the bits of $e_1$ and $e_2$ |
| hex:$n$ | = | the hex value interpreted as a bit-vector of $n$ bits |

**Isla traces and expressions** These are the traces returned by Isla and the symbolic expressions within them. An Isla trace describes the effects of a single instruction. Figure 2.7a shows a simplified trace of the `str` instruction from Figure 2.2b. The trace consists of SMT commands:

- `declare-const <var> <typ>` — declare a new variable `var` of type `typ`,
- `define-const <var> <exp>` — define a new variable `var` equal to `exp`,
- `assert <exp>` — assert `exp` is true,

and effects (some arguments omitted for brevity):

- `read-reg <reg> <value>` — read register `reg` and declare the result is equal to `value`,
- `read-mem <value> <addr> <b>` — read `b` bytes from address `addr` and declare the result is equal to `value`,
- `write-reg <reg> <value>` — write `value` to register `reg`,
- `write-mem <addr> <value> <b>` — write `value` to address `addr` (`b` bytes).

**Instruction traces and expressions** Isla traces are translated into a simpler representation that is easier to work with in the later stages of the pipeline. Again, each trace describes the effects of a single instruction. Figure 2.7b shows the instruction trace obtained by translating the Isla trace in Figure 2.7a. It has only four kinds of commands (also referred to as events):

| | | |
|---|---|---|
| $r \in$ | Register names | |
| $id_r, id_{nd} \in$ | $\mathbb{N}$ | Unique identifiers of a read and non-determinism variables |
| $sz \in$ | $\mathbb{N}$ | Data size in bytes |
| $e \in$ | Symbolic expressions over $var$ | |
| $event ::=$ | $\mathsf{WriteReg}(r, e)$ | Write $e$ to register $r$ |
| \| | $\mathsf{ReadMem}_{sz}(e_a, id_r)$ | Read a block of size $sz$ from address $e_a$, the result is represented by the read variable $id_r$ |
| \| | $\mathsf{WriteMem}_{sz}(e_a, e_v)$ | Write $e_v$ to a block of size $sz$ at address $e_a$ |
| \| | $\mathsf{Assert}(e)$ | Assert $e$ is true |

and there are three different kinds of variables used inside the symbolic expressions:

| | | |
|---|---|---|
| $var ::=$ | $\mathsf{Register}(r)$ | The value of the register $r$ at the beginning of the trace |
| \| | $\mathsf{Read}_{sz}(id_r)$ | The result of a memory reading operation, of size $sz$ |
| \| | $\mathsf{NonDet}_{sz}(id_{nd})$ | Variable representing non-determinism in the ISA spec, of size $sz$ |

Notice that there is no register read command, instead a variable $\mathsf{Register}(r)$ is used to denote the value of register $r$ before executing the instruction.

**Memory traces and state expression** The variables in the **instruction expressions** are local to the execution of a single instruction. On the other hand, **state expressions** contain variables bound to the full symbolic execution context.

To apply an **instruction trace** to a state, the **instruction variables** must be substituted by the appropriate **state expressions**. For example, a variable Register($r$) would be substituted by the symbolic expression stored in the register $r$ in the current state. Some of the variable types used in **state expressions** are shown below:

$$
\begin{aligned}
id_s &\in \mathbb{N} && \text{Unique identifier of a state} \\
id_r &\in \mathbb{N} && \text{Unique identifier of a read variable in a particular state} \\
var ::= \ & \mathrm{ReadVar}_{sz}(id_s, id_r) && \text{The result of a read in the state } id \\
\mid \ & \mathrm{Arg}(n) && \text{The } n\text{-th function argument} \\
\mid \ & \mathrm{RetAddr} && \text{The return address} \\
\mid \ & \ldots
\end{aligned}
$$

The memory in a given state is described using memory traces like the one shown in Figure 2.8. These consist of only two kinds of events:

$$
\begin{aligned}
e &\in \text{Symbolic expressions over } var \\
event ::= \ & \mathsf{Read}_{sz}(e_a, var) && \text{From } e_a\text{, read } var \text{ of size } sz \\
\mid \ & \mathsf{Write}_{sz}(e_a, e) && \text{To } e_a\text{, write } e \text{ of size } sz
\end{aligned}
$$

### State representation

The symbolic execution progresses from some initial state, constructing a state tree. Each symbolic state is represented using:

- Register map — **state expressions** describing the value in each register,
- Memory — represented using **memory traces**,
- Asserts — the path conditions for the state (i.e., the set of constraints that need to be satisfied for a program to reach this state).

An **instruction trace** is executed on a state to obtain a new state. The trace events are performed in sequence, modifying the state:

**WriteReg** updates the value in the register map.

**Assert** inserts a new path condition into the state's asserts.

**WriteMem** appends a Write entry to the **memory trace**.

**ReadMem** appends a Read entry to the **memory trace**, and creates a new ReadVar variable to represent the result. Sometimes, we can avoid using a ReadVar if the result can be determined from previous writes. This is achieved through memory caching.

**Memory caching**

| | address | value | |
|---|---|---|---|
| $\mathsf{Write}_8$ ( | 0x410fe8, | 0x0 | ) |
| $\mathsf{Write}_8$ ( | 0x410ff0, | 0x1 | ) |
| $\mathsf{Read}_8$ ( | 0x410fe8, | $\mathrm{ReadVar}_8(1, 0)$ | ) |
| $\mathsf{Write}_8$ ( | $\mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8$, | 0x2 | ) |
| $\mathsf{Read}_8$ ( | 0x410fe8, | $\mathrm{ReadVar}_8(1, 1)$ | ) |

**Figure 2.9:** Memory trace

Read-dwarf caches writes to substitute previously written values for read variables. After each write, all cache entries whose addresses could overlap with the new write are removed and a new entry is added. Consider the memory trace in Figure 2.9. The cache after the first two writes has the following entries:

$$0\mathrm{x}410\mathrm{fe}8 \mapsto 0\mathrm{x}0 \text{ (8 bytes)}$$
$$0\mathrm{x}410\mathrm{ff}0 \mapsto 0\mathrm{x}1 \text{ (8 bytes)}$$

For the next read (from 0x410fe8), the cache is used to determine that $\mathrm{ReadVar}_8(1, 0) = 0\mathrm{x}0$. After the next write (to $\mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8$), we need to remove both cache entries, because without any further context, we cannot determine if any of the following equalities hold:

$$0\mathrm{x}410\mathrm{fe}8 \stackrel{?}{=} \mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8$$
$$0\mathrm{x}410\mathrm{ff}0 \stackrel{?}{=} \mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8$$

For the next read (from 0x410fe8), we are not able to substitute a value for $\mathrm{ReadVar}_8(1, 1)$ because the cache is empty.

The memory can also be split into multiple mutually exclusive fragments. Read-dwarf keeps track of what fragment each pointer points into - its provenance. It is assumed that pointers with different provenance do not alias (point to the same address). By default, read-dwarf uses one fragment for the current stack frame and one for the rest of the memory. Every pointer derived from the stack pointer has **Stack** provenance, everything else **Main** provenance. The memory trace and cache, split between the two fragments, is shown in Table 2.2.

| | Main fragment | Stack fragment |
|---|---|---|
| **Trace** | $\mathsf{Write}_8(0\mathrm{x}410\mathrm{fe}8, \quad 0\mathrm{x}0)$<br>$\mathsf{Write}_8(0\mathrm{x}410\mathrm{ff}0, \quad 0\mathrm{x}1)$<br>$\mathsf{Read}_8(0\mathrm{x}410\mathrm{fe}8, \quad \mathrm{ReadVar}_8(1, 0))$<br>$\mathsf{Read}_8(0\mathrm{x}410\mathrm{fe}8, \quad \mathrm{ReadVar}_8(1, 1))$ | $\mathsf{Write}_8(\mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8, 0\mathrm{x}2)$ |
| **Cache** | $0\mathrm{x}410\mathrm{fe}8 \mapsto 0\mathrm{x}0$<br>$0\mathrm{x}410\mathrm{ff}0 \mapsto 0\mathrm{x}1$ | $\mathrm{Register}(\mathrm{SP}, 0) + 0\mathrm{x}8 \mapsto 0\mathrm{x}2$ |

**Table 2.2:** Memory trace split between fragments

## 2.5 Requirement Analysis

In my project proposal, I outlined two main goals and an extension:

- Enable symbolic execution of relocatable files, using symbols to represent the unknown addresses of sections.

- Enable (symbolically) evaluating the debug information in relocatable files.
- (extension) Create an automatic verification tool for semantic relocatability of ELF files.

All of these only concern ELF files compiled from C for AArch64. For the core part, I shall demonstrate that it can handle C programs compiled with no optimizations, containing any of the primitives in Table 2.3.

| **Primitive** | **Notes** |
|---|---|
| Control flow primitives | conditionals, loops, function calls |
| Global variables | reading, writing |
| Pointers | including when affected by relocations (e.g. pointing to global variables) |
| Arrays | |
| Structs | |

**Table 2.3:** C primitives to be supported

### 2.5.1 Work plan

I split the work to be done into the following work-items.

**Interpreting relocation entries** Create a representation that describes the effects of relocations, and convert relocation entries into this representation, according to the specification. This representation will be used by the later parts of the project.

**Relocation-aware DWARF** Modify linksem to process DWARF in relocatable files. While parsing the DWARF sections, process the relocation entries to insert symbolic values that represent the result of each relocation. Modify the functions that further process the DWARF to operate on symbolic instead of concrete values.

**Symbolic execution** Modify read-dwarf to allow symbolically executing relocatable files. Introduce new symbolic variables to represent the addresses of sections, and modify the execution pipeline to use them when executing instructions affected by relocations.

Use the interpreted debug information to visualize the result of the symbolic execution, displaying the values of variables.

**Semantic relocatability (extension)** Verify semantic relocatability of functions in an ELF file by constructing a bisimulation between two runs of the same function, but with possibly different addresses of sections. This will be done incrementally, first targetting simple functions and progressing to more complex ones.

### 2.5.2 Languages Used

Since most of the code was written inside existing codebases, I had to use the programming languages used by them. Most of the implementation was done inside read-dwarf, written in OCaml. I was familiar with OCaml from the Tripos, but had not had any further experience with it.

Modifications to linksem had to be written in a custom language Lem. Learning the syntax was simple, since it is almost identical to OCaml. One difficulty of developing in Lem was the

lack of editor support. I was unable to use autocomplete or any static analysis and refactoring features that exist for mainstream languages in most editors.

Some modifications were also done to Isla, written in Rust. I had previous experience developing in Rust, so this was without issues.

### 2.5.3 Development methodology

I used the agile development model. The project was initially divided into four main parts (described in §2.5.1), and further broken down into smaller subtasks. Each completed component was tested to inform the next steps. Converging on the final design required experimentation and iteration. I used git for version control, which allowed me to easily revert to previous versions of the code when needed. All code was backed up on GitHub.

# 3 | Implementation

This chapter is split into six sections:

- §3.1 describes a representation of relocations that is used in the rest of the project, and how it is constructed.
- §3.2 describes the modifications to linksem that allow reading the debug information from relocatable files.
- §3.3 describes the changes in read-dwarf's symbolic execution pipeline that allow executing relocatable files.
- §3.4 follows from the previous sections, describing how a complete relocatable program is executed. A script is implemented that symbolically executes a program and prints the execution path with evaluated debug information.
- §3.5 refines the concept of semantic relocatability and describes a tool implemented to verify it.
- §3.6 gives a repository overview.

## 3.1 Relocations

This section is about representing the relocation entries in a way that directly expresses the effect of the relocation. This representation will be used in multiple parts of the project, for processing relocations in DWARF and during symbolic execution.

### 3.1.1 Relocation types

As mentioned in §2.1.2, there are different processor-specific relocation types that describe what action must be performed. These actions are described in the ABI specification for each processor architecture. As we saw in Table 2.1, the actions typically consist of the following steps:

1. Evaluate a relocation operation.
2. Perform safety checks, such as an overflow check.
3. Extract a range of bits from the result.
4. Write the result into the target field at the relocation position. This is usually an immediate field of some instruction or a certain number of consecutive bytes in memory.

I chose a representation that expresses these four steps. The translation of Table 2.1 into this representation is shown in Table 3.1. It is a record type consisting of four fields:

- Operation — the AST of the relocation operation. It is formed by simple arithmetic operators (Plus, Minus) and special operators from the ABI documentation (Page($x$)). The set of operators (as a type) has already been defined in linksem.
- Safety Checks — I only needed to consider two kinds of safety checks:
  - Overflow($min$, $max$) — check that the value is between $min$ (inclusive) and $max$ (exclusive) value,
  - Alignment($n$) — check that the bottom $n$ bits are zero (important for aligned read and write operations).

  This field contains a list of these checks.
- Mask — a pair $(hi, lo)$ identifying the range of bits to be extracted.
- Target Field — I made this field generic, because every architecture has a different set of relocation targets. For each supported architecture (currently only AArch64), I created a sum type consisting of the possible fields.

| **Name** | **Representation** | | | |
|---|---|---|---|---|
| | **Operation** | **Safety Checks** | **Mask** | **Target** |
| R_AARCH64_ADR_PREL_PG_HI21 | Page(S+A) - Page(P) | Overflow($-2^{32}, 2^{32}$) | (32,12) | ADRP |
| R_AARCH64_LDST64_ABS_LO12_NC | S + A | Alignment(3) | (11,3) | LDST 3 |
| R_AARCH64_ABS64 | S+A | — | (63, 0) | Data64 |

**Table 3.1:** Representation of selected AArch64 relocations

The complete list of supported relocation types is shown in Appendix A.

### 3.1.2 Problems with module dependencies

The way the linksem repository is organized prevents using the above representation in some places. The repository consists of two main modules, for ELF and the ABI. The ELF module contains the core specification of ELF files, including DWARF. The ABI module contains all machine-specific logic which includes relocations. Code in the ABI module can depend on the ELF module, but not the other way around.

The issue is that some relocations occur inside DWARF, and in order to interpret DWARF, it is necessary to interpret the relocations. To accommodate this dependency, I implemented a universal representation of relocations that removes all the ABI-specific dependencies and resides in the ELF module.

### 3.1.3 Universal representation

The universal representation consists of the same fields as the normal ABI-aware representation, except for the **operation**. Instead, the operation is evaluated into a symbolic expression, consisting of elementary arithmetic and bit operations, with symbols representing the load address of each ELF section. I implemented the evaluation function inside the ABI module. An example is shown in Table 3.2.

| Operation | Resulting symbolic expression |
|---|---|
| S | Section(.data)+0 |
| A | 0 |
| P | Section(.text)+4 |
| S+A | Section(.data)+0+0 |
| Page(S+A) - Page(P) | ((Section(.data)+0+0) & ∼0xFFF) - ((Section(.text)+4) & ∼0xFFF) |

**Table 3.2:** Translation of relocation operations

To handle the ABI-specific **target fields**, I created a new type to represent common target fields, which are needed by DWARF. These are (my naming):

- `Data32` — 4 consecutive bytes starting at the specified location,
- `Data64` — 8 consecutive bytes starting at the specified location.

Each architecture's implementation defines a partial mapping from its full set of target fields to one of these two data fields.

### 3.1.4 Obtaining relocations for a section

First, the section that contains the relocation entries is found, and its content is parsed (using a preexisting function) into a list of relocation entries. Next, each relocation entry is interpreted according to the ABI specification into its universal representation.

This process is implemented as a function taking three arguments. A preprocessed ELF file, a relocation interpreter, and a name of the section. The relocation interpreter is a function from a relocation entry to a universal representation (also requiring context like the symbol table and section ID). Relocation interpreters are architecture-specific, thus they are defined in the ABI module.

## 3.2 Relocation Aware DWARF

As we saw in §2.1.3, DWARF contains memory address references that are inserted by relocations. In relocatable files, the concrete memory addresses are not yet defined, but we can describe them using symbolic expressions (see Figure 3.1).

```
<0><b>: DW_TAG_compile_unit
   <12>   DW_AT_name        : example.c
   <1e>   DW_AT_low_pc      : .text+0x0
   <26>   DW_AT_high_pc     : 0xc
   <1><2a>: DW_TAG_variable
      <2b>   DW_AT_name        : x
      <35>   DW_AT_location    : DW_OP_addr .data+0x0
   <1><46>: DW_TAG_subprogram
      <47>   DW_AT_low_pc      : .text+0x0
      <4f>   DW_AT_high_pc     : 0xc
      <55>   DW_AT_name        : f
      <2><5b>: DW_TAG_formal_parameter
         <5c>   DW_AT_location    : DW_OP_reg0
         <5e>   DW_AT_name        : a
```

**Figure 3.1:** DWARF debug information with symbolic values (some attributes omitted)

I modified the functions for parsing DWARF in linksem to obtain this kind of representation. Linksem also provides analysis functions to extract specific information from DWARF, such as the list of all variables and their locations. I modified these functions to operate on symbolic values. Table 3.3 shows the result of the variable location analysis.

| variable | type | low pc address | high pc address | location |
|---|---|---|---|---|
| x | int | .text+0x0 | .text+0xc | DW_OP_addr .bss+0x0 |
| a | int | .text+0x0 | .text+0xc | DW_OP_reg0 |

**Table 3.3:** Variable locations

Linksem parses DWARF by extracting the bodies of relevant `.debug` sections, which are then interpreted by a series of parsing functions.

For relocatable files, the section bodies are extracted together with relocation entries to construct a **symbolic byte sequence**. Reading from a symbolic byte sequence yields either a concrete value or a symbolic value if that place is targeted by a relocation. The symbolic values are represented by newly introduced **symbolic types**.

### 3.2.1 Symbolic types

Linksem represents numeric values using `natural` and `integer` types. I implemented their symbolic versions `sym_natural` and `sym_integer`. Notice that all symbolic expressions in Figure 3.1 and Table 3.3 have the form "section plus a constant". This is expected because they refer to a specific location in some section. To keep the implementation simple, I chose a representation that restricts the expressions to this form. If a case was found where a more complex symbolic expression is needed, the representation could be changed. I have not found such a case during testing.

Both types use the same underlying representation illustrated in Figure 3.2. Some operations with them are shown in Figure 3.3. Each operation raises an exception if the result cannot be simplified to one of the two options, indicated by $\perp$. The exceptions contain descriptive error messages to help with debugging and discovering new cases that need to be covered.

$$
\begin{aligned}
n &\in \mathbb{Z} \\
s &\in \text{Section names} \\
\mathit{value} ::= \ & \text{Absolute}(n) && \text{Representing value } n \\
\mid \ & \text{Offset}(s, n) && \text{Representing value } v_s + n \\
& && \text{where } v_s \text{ is a symbol representing} \\
& && \text{the load address of section } s
\end{aligned}
$$

**Figure 3.2:** Representation of DWARF symbolic values

$$\begin{aligned}
\mathrm{Abs}(n_1) + \mathrm{Abs}(n_2) &= \mathrm{Abs}(n_1 + n_2) \\
\mathrm{Off}(s_1, n_1) + \mathrm{Abs}(n_2) &= \mathrm{Off}(s_1, n_1 + n_2) \\
\mathrm{Abs}(n_1) + \mathrm{Off}(s_2, n_2) &= \mathrm{Off}(s_1, n_1 + n_2) \\
\mathrm{Off}(s_1, n_1) + \mathrm{Off}(s_2, n_2) &= \bot
\end{aligned}$$

**(a)** Addition

$$\begin{aligned}
\mathrm{Abs}(n_1) - \mathrm{Abs}(n_2) &= \mathrm{Abs}(n_1 - n_2) \\
\mathrm{Off}(s_1, n_1) + \mathrm{Abs}(n_2) &= \mathrm{Off}(s_1, n_1 - n_2) \\
\mathrm{Abs}(n_1) - \mathrm{Off}(s_2, n_2) &= \bot \\
\mathrm{Off}(s_1, n_1) - \mathrm{Off}(s_2, n_2) &= \begin{cases} \mathrm{Abs}(n_1 - n_2) & \text{if } s_1 = s_2 \\ \bot & \text{if } s_1 \neq s_2 \end{cases}
\end{aligned}$$

**(b)** Subtraction

**Figure 3.3:** DWARF symbolic operations (writing Abs and Off instead of Absolute and Offset for compactness)

### 3.2.2 Symbolic byte sequence

To keep the implementation simple, we only allow reading:

(a) any number of bytes unaffected by relocations, or

(b) the exact number of bytes affected by some relocation.

With this restriction, a symbolic byte sequence can be represented as concrete byte sequences interleaved with symbolic segments represented using `sym_natural` (Figure 3.4).

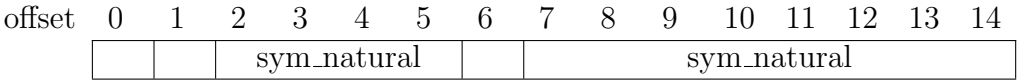



**Figure 3.4:** Symbolic byte sequence

To avoid having to deconstruct the byte sequence containing the body of a section and interleave it with the symbolic parts, I kept the original byte sequence as is and added a mask representing the symbolic bits (Figure 3.5).

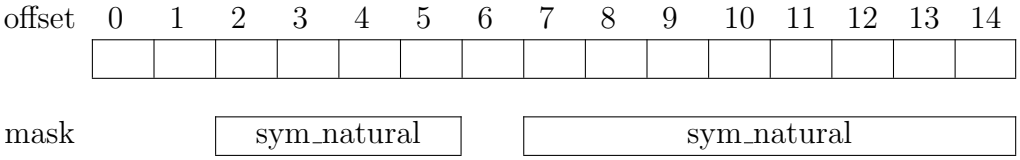



**Figure 3.5:** Symbolic byte sequence using a mask

The mask is represented as a list of mask entries of two variants:

- NoSym($n$) — signals that the next $n$ bytes are not symbolic and should be taken from the underlying raw byte sequence.
- SymVal($n$, $v$) — signals that the next $n$ bytes should be treated as containing the symbolic value $v$.

This representation was chosen because it makes reading and removing bytes from the beginning (an extremely common operation) efficient.

Originally, linksem uses functions that read a varying number of bytes from a byte sequence (1, 2, 4, ...), returning them as a tuple. Corresponding functions were implemented for the symbolic byte sequence, with the following return type:

```
type read_result 'a =
| Bytes of 'a
| Sym of sym_natural
```

Here, `'a` ranges over byte tuples of different lengths. The two cases are handled by the caller — a low-level parsing function. Some functions (e.g., when parsing a string) only expect `Bytes`, while others (when parsing a number) handle both options.

### 3.2.3 Constructing a symbolic byte sequence

To construct a symbolic byte sequence for a section, the raw bytes of the section are extracted from the ELF file and the mask is constructed using relocation entries. The symbolic expression from the universal representation of a relocation is converted to the simplified form of `sym_natural` and the relocation target (only `Data32` and `Data64` should appear here) is used to determine the width of the respective mask entry. The safety checks and mask have no effect in the simple relocations appearing in DWARF.

Then, the usual DWARF-reading code is used, modified to use the symbolic types and symbolic operations. The functions for evaluating DWARF expressions and analysis functions are modified in the same way.

## 3.3 Read-dwarf Symbolic Execution

This section describes the changes I made in read-dwarf to allow symbolically executing relocatable files. The first step was modifying the types used to represent parts of the ELF file and the functions used to construct them from the output of linksem. Next, I modified the symbolic execution pipeline.

I started with the process of retrieving instruction traces. Changes were needed on the downwards path — propagating the relocation information to Isla, as well as the upwards path — processing the traces. I followed the pipeline, making changes as needed.

Lastly, changes had to be made in the state representation, and the process of executing traces. This included methods for simplifying symbolic expressions.

Since each part is strongly tied to the pre-existing implementation of read-dwarf, I include **Context** paragraphs to give the necessary background.

### 3.3.1 Retrieving ELF symbols

**Context** Read-dwarf uses linksem to extract data from the ELF file. It extracts a set of symbols, each with its address and the data from that address (for global variables, this is their initial value, for functions, it is their code). Read-dwarf constructs a symbol table, allowing look-up of symbols by name and by address.

The addresses were changed from absolute (represented by an integer) to section-relative, represented by a record type `{ section : string; offset : int }`. The symbol table by address also had to be modified into a collection of tables per section.

The symbol data, originally a byte sequence (`ByteSeq.t`), was changed to a record type `{ data : BytesSeq.t; relocations : Relocations.t }`. The `Relocations.t` is a map from offsets to relocations in the universal format, consisting of the symbolic relocation value, safety checks, mask and the target field.

The relocations are extracted for each section using the function I implemented in linksem, and we filter those that apply to each symbol. After the symbols are loaded, read-dwarf looks up the requested function in the symbol table, and starts executing it.

### 3.3.2 Execution pipeline

The rest of the section explains how the execution pipeline was modified, to enable executing relocatable functions. I will demonstrate this using the example in Figure 3.6, mostly focusing on the execution of the `str` (memory write) instruction (marked in the figure).

```
  0:  90000008  adrp  x8, 0
  0:  R_AARCH64_ADR_PREL_PG_HI21    x
> 4:  f9000100  str   x0, [x8]
  4:  R_AARCH64_LDST64_ABS_LO12_NC  x
  c:  d65f03c0  ret
```

**Figure 3.6:** Objdump of a relocatable file (repeated from Figure 2.2a), with the relevant instruction marked

After executing this function, we should see a memory trace with

$$\mathsf{Write}_8(\mathsf{Section}(.\mathrm{data}), \mathsf{Arg}(0))$$

Subsections 3.3.3, 3.3.4, 3.3.5 cover the process of obtaining an instruction trace, and Subsections 3.3.6, 3.3.7 cover applying the trace to the state.

### 3.3.3 Obtaining the Isla trace

**Context** Read-dwarf obtains the Isla trace by sending a request (e.g. `execute #xf907f500`) to Isla-client. The response is parsed using Isla-lang (a parsing library for Isla output), and the trace is simplified by removing unnecessary parts.

Isla has a feature that enables symbolic opcodes with missing bits represented by *segment variables*. For example, `#b1111100100000 x0:9 #b0100000000` has 9 missing bits represented by `x0`. I used this feature to process instructions affected by relocations. A variable is used to represent the bits overwritten by the relocation.

#### Forming a request

The relocation target determines which bits does the relocation write to. This is used to form the symbolic opcode as shown in Table 3.4. Table 3.5 shows a concrete example for the `str` instruction. The request is sent to Isla-client.

| Relocation target | Symbolic opcode | Segments |
|---|---|---|
| ADRP | opc[31:31]␣**x0:2**␣opc[24:28]␣**x1:19**␣opc[0:4] | x0 = bits [0:1]<br>x1 = bits [2:20] |
| LDST $n$ | opc[(22-$n$):31]␣**x0:(12-$n$)**␣opc[0:9] | x0 = bits [0:11 − $n$] |

**Table 3.4:** Forming the symbolic opcode for selected relocation targets. Ranges of bits from the raw opcode are denoted as opc[*lo*:*hi*]. The last column shows which symbolic segments correspond to which bits of the relocation value.

| Raw opcode | Relocation target | Symbolic opcode |
|---|---|---|
| #b1111100100000000000000100000000 | LDST 3 | #b1111100100000 x0:9 #b0100000000 |

**Table 3.5:** Example of forming a symbolic opcode

### Isla-client

While Isla internally supports symbolic opcodes, this feature was missing in Isla-client and had to be added. Only minor changes were needed — first, to parse the symbolic opcode from the request; and second, to emit an additional `segments` message that maps segment variables to Isla variables used in the trace. Figure 3.7 shows the communication between read-dwarf and Isla-clent when executing the `str` instruction.

```
[read-dwarf]: execute #b1111100100000 x0:9 #b0100000000
[isla]:       Segments
                (segments
                  (|x0| 9 v0))
[isla]:       StartTraces
[isla]:       Trace (normal)
                (trace
                  (declare-const v0 (_ BitVec 9))
                  (declare-const v1 (_ BitVec 64))
                  (declare-const v2 (_ BitVec 64))
                  (read-reg |R8| v1)
                  (read-reg |R0| v2)
                  (define-const v3
                    ((_ extract 51 0 )
                      (bvadd
                        v1
                        (concat x0000000000 (concat v0 #b000))
                      )
                    )
                  )
                  (define-const v4 ((_ zero_extend 12) v3))
                  (write-mem v4 v1 4)
                )
[isla]:       EndTraces
```

**Figure 3.7:** Isla request and response with symbolic instruction opcode. The `segments` entry describes that the segment variable `x0` is mapped to the variable `v0`

### Parsing and simplification

The response is parsed as before, except now with the additional "segments" entry. For that, an appropriate parser from Isla-lang is used.

Some difficulties were caused by the trace simplification. After the trace is pruned of unnecessary commands, variables are renamed to be sequential (`v0`, `v1`, ...). This causes a problem when a renamed variable was mapped to a segment. This had a simple fix: disallow renaming variables that are mapped to segments (these are always the first $k$ variables, so everything stays neat). The difficult part was diagnosing the problem, without an extensive knowledge of the codebase.

### 3.3.4 Trace processing

**Context** At this stage, **Isla trace** is converted to an **instruction trace**. An important aspect of this conversion is substituting **Isla variables** by the appropriate **instruction variables**.

As the trace is being processed, a mapping from Isla variables to the new variables is kept and used to substitute variables in all expressions.

A new **instruction variable** kind was added:

$$\begin{aligned} var ::= \ & \ldots \\ & \mid \text{Segment}(s, n) \qquad \text{A } n \text{ bit wide symbolic segment in the opcode, named } s \end{aligned}$$

to represent each symbolic segment in the instruction's opcode.

This is then substituted for Isla variables according to the "segments" entry in the Isla response.

Figure 3.8 shows the **instruction trace** (consisting of just one event) we obtain from the **Isla trace** in Figure 3.7. The variable `v0` was substituted by Segment(x0, 9). Notice that the Segment(x0, 9) variable appears in the address expression, indicating it is affected by relocation.

$$\mathsf{WriteMem}_8(\text{Register}(\text{R8})[0:51] + \text{0x0:40}.\text{Segment}(\text{x0}, 9).\text{0x0:3}, \quad \text{Register}(\text{R0}))$$

**Figure 3.8:** Instruction trace with segments

### 3.3.5 Caching

**Context** Obtaining a trace for an instruction is computationally expensive, therefore the traces (both **Isla traces** and **instruction traces**) are cached to speed up successive runs.

The cache was originally indexed by the instruction opcode. For instructions affected by relocations, we use the raw opcode and the relocation target as a key.

The caching is also one reason why the Segment variables are used in **instruction trace** instead of substituting them by the appropriate (symbolic) bits of the relocation values. Otherwise, the relocation value would need to be included in the caching key. This way, instructions differing only by their relocation value can benefit from a single cache entry.

### 3.3.6 Trace execution

**Context** An **instruction trace** is executed by performing the events it consists of and updating the state. During the execution, the **instruction expressions** must be expanded into **state expressions**. This is done by substituting **instruction variables** by the appropriate **state expressions**. The same has to be done for the new Segment variables.

At this stage, the Segment variables are substituted by the appropriate bits of the relocation values. The value to be substituted is constructed from the universal representation of the relocation and the segment mapping shown in Table 3.4.

Consider our `str` example with the relocation `R_AARCH64_LDST64_ABS_LO12_NC x`. The universal representation of this relocation is shown in Table 3.6.

| Value | Safety Checks | Mask | Target |
|---|---|---|---|
| Section(.data)+0+0 | Alignment(3) | (11,3) | LDST 3 |

**Table 3.6:** Universal representation of `R_AARCH64_LDST64_ABS_LO12_NC x`, when `x` is located at .data+0.

First, we take the relocation value and apply a mask, resulting in

$$(\text{Section}(.\text{data}) + 0 + 0)[3:11]$$

Then, referring to the last column of Table 3.4, we obtain an expression for each symbolic segment (in this case the operation is trivial)

$$\text{x0} = (\text{Section}(.\text{data}) + 0 + 0)[3:11][0:8]$$

or in a simplified form

$$\text{x0} = \text{Section}(.\text{data})[3:11]$$

This is the expression that will be substituted for variable $\text{Segment}(\text{x0}, 9)$.

We also construct expressions representing the safety checks. We will call them *relocation assertions*, and they will be used to simplify expressions (see §3.3.8). In this case, Alignment(3) translates to

$$\text{Section}(.\text{data})[0:2] = 0\text{x}0{:}3$$

To accommodate these expressions I added a new kind of **state variable**:

$$\begin{aligned} var ::= {} & \ldots \\ & \mid \text{Section}(s) \qquad \text{The address of section } s \end{aligned}$$

Table 3.7 demonstrates the substitution in the address expression in the trace from Figure 3.8. We arrive at an expression corresponding to the address of `x`.

| | |
|---|---|
| Registers | $\text{R8} \mapsto \text{Section}(.\text{data})[12:63].0\text{x}0{:}12$ |
| Segments | $\text{x0} \mapsto \text{Section}(.\text{data})[3:11]$ |
| Original **instruction expression** | $\text{Register}(\text{R8})[0:51] + 0\text{x}0{:}40.\text{Segment}(\text{x0}, 9).0\text{x}0{:}3$ |
| Expanded **state expression** | $(\text{Section}(.\text{data})[12:63].0\text{x}0{:}12)[0:51]$ $+0\text{x}0{:}40.(\text{Section}(.\text{data})[3:11]).0\text{x}0{:}3$ |
| Simplified **state expression** | $(\text{Section}(.\text{data}))[0:51]$ |

**Table 3.7:** Example of expanding an instruction expression

### 3.3.7 Memory operations

**Context** Memory is represented as a trace of read and write operations, with read variables representing the results of each read. In addition, a cache is used to keep track of previously written values, so we can substitute them for read variables if we can guarantee that they were not overwritten. The memory can be split into multiple mutually exclusive fragments, to prevent aliasing problems.

Symbolic execution can produce a memory trace accessing multiple sections, as shown in Figure 3.9. The naive caching mechanism is unable to determine that the addresses Section(.data) and Section(.bss) are distinct and substitute the read variable accordingly.

| | **address** | **value** | |
|---|---|---|---|
| $\mathsf{Write}_8$ ( | Section(.data), | 0x0 | ) |
| $\mathsf{Write}_8$ ( | Section(.bss), | 0x1 | ) |
| $\mathsf{Read}_8$ ( | Section(.data), | $\mathrm{ReadVar}_8(1, 0)$ | ) |

**Figure 3.9:** Memory trace

I solved this by assigning each section into a separate memory fragment. When performing a memory operation (not stack-relative), the address expression is simplified into the form section-plus-offset (see §3.3.8). The section is then used to pick the correct fragment. The memory traces in each fragment are illustrated in Figure 3.10.

| **.data fragment** | **.bss fragment** |
|---|---|
| $\mathsf{Write}_8$(Section(.data), 0x0) | $\mathsf{Write}_8$(Section(.bss), 0x1) |
| $\mathsf{Read}_8$(Section(.data), $\mathrm{ReadVar}_8(1, 0)$) | |

**Figure 3.10:** Memory trace split between section fragments

This approach has one downside: It forbids symbolic pointers for which we do not know what section they point into. When reading/writing through these pointers, we cannot determine which fragment to perform the operation in. The core of this project does not require these kinds of pointers, but for some later extensions, I used a single fragment for all sections. In that case, some read variables could not be eliminated.

### 3.3.8 Simplification

Consider a call to some function `f`, located at Section(.text.f), from Section(.text.main). We expect this to set the PC register to the symbolic address Section(.text.f). Instead, we end up with an expression like this

$$\text{Section(.text.main)} + F[\text{Section(.text.f)} - \text{Section(.text.main)}]$$

where $F[x]$ is an expression that does the following to $x$:

1. Extracts bits 2 to 27 inclusive.
2. Concatenates two zeros to the right.
3. Sign-extends to 64 bits.

This is because the call instruction takes a relative offset to the target function

$$\text{Section(.text.f)} - \text{Section(.text.main)}$$

without the two least significant bits, in a 26 bit immediate field. If the offset does not fit into the immediate field or is not a multiple of 4, it gets truncated as expressed by $F[x]$.

Notice that when the safety checks $\text{Overflow}(-2^{27},2^{27})$ and Alignment(2) are satisfied, there is no truncation and we can show $F[x] = x$. This allows us to simplify the expression to the expected form. The question is how to perform these simplifications automatically.

**Lazy approach** We delay the simplification until an expression is used as an address for a memory operation. At that point, we know the address can be expressed as Section$(s)+x$, where $x$ does not depend on any of the Section variables. In my implementation, I only considered $x$ being constant, since this was enough to execute complete programs, but in general, $x$ can also be symbolic.

First, $x$ is found by evaluating the expression with all Section variables set to zero. Next, we take all the Section variables contained in the expression as candidates for $s$. Finally, I used an SMT solver to check each candidate. I used the pre-existing utility functions to construct a query that checks if the set of safety checks implies that the candidate expression is equal to the original.

This approach leaves unsimplified expressions in the state, making it less interpretable, and requires keeping all safety checks around. To address these disadvantages, I introduced a second approach.

**Eager approach** We simplify the expression immediately after an **instruction expression** is converted to a **state expression** (§3.3.6). The challenge is that the expected simplified form is not always clear. Usually, the form is Section$(s)+x$, but there are exceptions. For example, the `adrp` instruction forms an address of a 4KB page, meaning the last 12 bits are zeroed. The expected form is (Section$(s)+x)[12:63]$.0x0:12.

Rather than hardcoding this, I used a more general approach. Notice that in the example, $F[x]$ is an expression, containing a subexpression $x$, it can be simplified to. This is a common pattern. We can perform the simplification automatically by trying all subexpressions and checking (using an SMT solver) if they are equal to $F[x]$ (under the assumptions imposed by the safety checks). To simplify the full expression, we perform this procedure on every subexpression. The complete algorithm is described in Algorithm 1. The set of relocation assertions (constructed from the safety checks) is used as the hypothesis ($hyp$).

**Algorithm 1** Simplification

**Require:** set of hypothesis $hyp$, expression $e$
**Ensure:** a simplified expression

```
function SIMPLIFY(hyp, e)
    e_new ← e
    for e' ∈ subexpressions(e) do                    ▷ try all sub-expressions
        if can prove hyp ⟹ e = e' then
            e_new ← e'
            break
    for e' ∈ children(e_new) do                      ▷ recursively simplify all child expressions
        e' ← SIMPLIFY(hyp, e')
    return e_new
```

## 3.4 Executing full programs

For this section, we only consider complete relocatable programs (with a main function), with no inputs and no side effects. I will show how such program is executed using the symbolic execution mechanism, and the result displayed. For a well-behaved program, we expect the state tree resulting from the symbolic execution (after removing all impossible paths) to form a line - no branching should depend on the symbolic addresses of sections.

### 3.4.1 State initialization

Before the symbolic execution can begin, the initial state needs to be prepared. This involves initializing the system registers, setting up memory fragments, and initializing other registers according to the ABI specification. Three additional steps were added:

1. Create memory fragments for each ELF section (recall §3.3.7).
2. Generate constraints about section's addresses, asserting that each section fits in memory, does not include the null address, is correctly aligned, and no two sections overlap. The symbolic executor can use them to simplify the path conditions and prune impossible execution paths.
3. Write the initial values of global variables (or any other objects) into the memory[1].

Step 3 is done by iterating over all object symbols in the symbol table. For each object, we write its content into memory, using the same mechanism as when executing an instruction trace. The object's content can also be affected by relocations that need to be handled.

C allows initializing global pointers with references to other symbols, as in Figure 3.11. Relocation is used to insert the correct pointer value. To construct the initial value, we need to substitute the part affected by the relocation with the corresponding symbolic value. The symbolic value is constructed by converting the relocation value and applying the mask (analogous to §3.3.6). The relocation target is used to determine the width of the part that is substituted. Only data relocations (recall `Data32` and `Data64` from §3.1.3) are allowed. If there are multiple relocations, they are processed sequentially.

```
int x;
int *p[] = {0, &x, 0};

int main() {
    ...
}
```

| | | |
|---|---|---|
| **Raw data** | 0x0:192 | |
| **Relocation** | `R_AARCH64_ABS64` `x` at offset +8 bytes | |
| | **value** | Section(.bss) |
| | **target** | Data64 (64 bits) |
| **Result** | 0x0:64.Section(.bss).0x0:64 | |

**Figure 3.11:** Global pointer initialization

### 3.4.2 Interpreting the result

After initializing the state, the symbolic execution is run and we obtain a symbolic state tree. If the tree consists of only a single execution path, we can print the sequence of instructions that were executed, and evaluate the debug information between them. This provides a convenient, human-readable view of the program's execution.

The format is designed for symbolic executions that do not branch on symbolic values, but can also handle those that do. The branches are printed after each other, and the user is alerted using a `BRANCH!` message. This usually suggests that the program is not semantically relocatable.

I used pre-existing functionality to print the instructions along with static debug information, such as the location in the C file each instruction corresponds to. In addition, I wrote code to obtain and print the values of the variables described in the debug information.

[1]Note that this step is only valid when executing the whole program from the start (the main function), when all global variables still contain their initial values. The read-dwarf's intended use case was symbolically executing individual functions, for which this would not be valid, therefore this step was not previously implemented.

### 3.4.3 Evaluating debug variables

Read-dwarf defines a simplified representation of DWARF. This includes a list of global variables and functions with their local variables. Each variable has a list of location descriptions, each with a range of PC addresses where it is valid. In DWARF, the locations are specified through (arbitrarily complicated) expressions, but read-dwarf uses pattern matching to identify four simple cases in Figure 3.8. If none of them are applicable, it is kept in the expression form.

| | |
|---|---|
| Register($r$) | In the register $r$ |
| RegisterOffset($r$,$o$) | At the address in register $r$ with an offset $o$ |
| StackFrame($o$) | At an offset $o$ from the base of the stack frame |
| Global($a$) | At the address $a$ |

**Table 3.8:** DWARF locations

To print the values of variables at a given execution point, I iterate through all global and local variables and check if the current PC address falls into the range of any of its location descriptions. If yes, I read the value from the state depending on the location description. Only the four simple variants are supported, with additional `Const` location I added to describe variables that are optimised into a constant. Only the original four were seen in programs compiled without optimisations.

More complicated location descriptions would require symbolically evaluating the DWARF expressions, operating on the read-dwarf's symbolic values. Considering the time constraints, this was not implemented.

## 3.5 Semantic relocatability

This section refines the concept of semantic relocatability and my implementation for its verification.

### 3.5.1 Theory

Consider a *relocated instance* of an object file to be the result of assigning a concrete address to each section and applying relocations. I will use the same term to refer to the machine state (memory and registers) when executing functions in a *relocated instance*.

For a well-behaved program, we expect each *relocated instance* to have the same behaviour. For full programs (a main function), we can define it as:

**Definition 3.5.1** *A relocatable program is semantically relocatable if for any given inputs, every relocated instance of the program produces the same output and side effects.*

This definition is about whole programs, but relocatable files usually do not contain full programs, only functions to be called by other binaries. To reason about their behaviour, it is useful to have a function-local definition.

**Definition 3.5.2** *A function is semantically relocatable if when it is executed in two relocated instances, such that all values reachable by the function are semantically equivalent between the two instances (see Definition 3.5.3), all externally observable effects (memory/register writes) will write semantically equivalent values.*

Reachability is defined by the C semantics. Only global variables, function arguments and values derived from them are reachable. Semantic equivalence is defined as follows:

**Definition 3.5.3** *Two values are semantically equivalent if one of the two holds:*

- *The values are pointers, and they point to the same offset in the same section.*
- *The values are base types (integers, floats, ...), and they have equal values.*

This notion naturally extends to composite types; for example, structs are semantically equivalent if all of their corresponding members are. The definition requires knowing the intended type of each value, which we assume to be the static C type encoded in DWARF.

### 3.5.2 Simulation

I verify the semantic relocatability of a function, by symbolically executing it and demonstrating a bisimulation between two copies of the symbolic execution tree (recall the procedure from §2.3), assuming the pre-conditions stated in Definition 3.5.2.

The bisimulation relation is simply a one-to-one map between the corresponding copies of each state. This approach only works for functions without unbounded loops, since they cause the symbolic execution to not terminate.

To verify the bisimulation, we need to check the equivalence between the path conditions of the corresponding states under the initial assumptions. We also need to check the semantic equivalence of the values written by the function and the return value.

### 3.5.3 Algorithm

Consider a pair of states we wish to verify the bisimulation condition for. First, we encode the initial assumptions inside a *verification context*. Then we iterate through the memory traces, updating the verification context and checking the semantic equivalence of the written values. We use the final verification context to prove the equivalence of path conditions and the semantic equivalence of return values. The verification context consists of three kinds of constraints.

**SMT constraints** ($C$) is simply a set of SMT expressions that are assumed to be true. They usually express equality between values in the two instances.

**Global memory constraints** ($G$) express the semantic equivalence between pointers. It is expressed as a set of triples $(a_1, \tau, a_2)$, denoted as $a_1 \approx_\tau a_2$, where $a_1, a_2$ are semantically equivalent pointers that point to values of type $\tau$. We also assume that all values derived from them (e.g. by dereferencing) are semantically equivalent.

**Stack constraints** ($S$) are used to generate constraints about values read from the stack. We only use them for the stack-passed function arguments. For the values that the function itself writes on the stack, we rely on the caching mechanism to read back the precise values. It is a map from stack locations to one of two equivalence kinds:

- Eq — the values are equal.
- Indirect$_\tau$ — the values are semantically equivalent pointers pointing to values of type $\tau$.

The location is described by a pair consisting of the offset from the stack base and a bytes length.

All pointers referenced by stack and global memory constraints must point **outside of the current stack frame**. This prevents aliasing problems where a value might have been overwritten by a stack write. My algorithm preserves this invariant.

At the beginning, we initialize the verification context accoding to the information about global variables and function arguments:

1. Create a global memory constraint for each global variable.

2. For each argument passed in a register, if it is a base type, we generate an SMT equality constraint between the values in said register. If it is a pointer, we generate a global memory constraint.

3. Generate stack constraints based on stack-passed arguments. If an argument is of a base type, it will be an Eq constraint, if a pointer, an Indirect constraint.

Afterwards, we iterate through the memory traces of the two instances in lockstep, updating the constraints. Each pair of memory operations should be of the same kind (read or write), otherwise the algorithm fails[2]. The constraints are updated each time according to what kind of memory operation it is.

**Stack read** If both reads are from the same stack offset, for which we have a stack constraint, we generate a corresponding constraint about the read values.

**Stack write** We erase all stack constraints about values that get overwritten by the operation.

**Global read** We look up a global memory constraint for the given pair of addresses and generate a corresponding constraint about the read values.

**Global write** We look up a global memory constraint for the given pair of addresses, and check if the newly written values still satisfy it.

The full set of inference rules is shown in Figure 3.12. The conclusions have a form

$$C, S, G \xrightarrow{\text{operation}_1,\text{operation}_2} C', S', G'$$

where $C, S, G$ represent the SMT, Stack and Global memory constraints respectively, before processing the pair of memory operations $(\text{operation}_1, \text{operation}_2)$, and $C', S', G'$ are the constraints after processing the operations.

I use $C \vdash P$ to denote that a predicate $P$ is provable from the expressions in $C$. This is needed when the predicate $P$ contains symbolic values. When writing $C \vdash x_1 \approx_\tau x_2 \in G$, this essentially means

$$\exists (x_1' \approx_\tau x_2') \in G.\ C \vdash x_1 = x_1' \wedge x_2 = x_2'$$

When introducing a global memory constraint, we also create SMT constraints about the pointer values. These express that both pointers are in a valid memory range and either both or none are null. The set of these constraints is

$$\mathrm{Ptr}(v_1, v_2) = \begin{Bmatrix} 0 \leq v_1 \leq \text{max\_address} \\ 0 \leq v_2 \leq \text{max\_address} \\ v_1 = 0 \Leftrightarrow v_2 = 0 \end{Bmatrix}$$

[2] Always the case when processing two copies of the same execution tree.

$$\frac{o \text{ is not symbolic} \quad S(o, sz) = \text{Eq}}{C, S, G \xrightarrow{\text{StackRead}_{sz}(o,v_1),\text{StackRead}_{sz}(o,v_2)} C \cup \{v_1 = v_2\}, S, G} \text{(StackReadEq)}$$

$$\frac{o \text{ is not symbolic} \quad S(o, sz) = \text{Indirect}_\tau}{C, S, G \xrightarrow{\text{StackRead}_{sz}(o,v_1),\text{StackRead}_{sz}(o,v_2)} C \cup \text{Ptr}(v_1, v_2), S, G \cup \{v_1 \approx_\tau v_2\}} \text{(StackReadIndirect)}$$

$$\frac{o \text{ is not symbolic} \quad (o, sz) \notin \text{dom}(S)}{C, S, G \xrightarrow{\text{StackRead}_{sz}(o,v_1),\text{StackRead}_{sz}(o,v_2)} C, S, G} \text{(StackReadNone)}$$

$$\frac{o \text{ is not symbolic}}{C, S, G \xrightarrow{\text{StackWrite}_{sz}(o,v_1),\text{StackWrite}_{sz}(o,v_2)} C, S \setminus \{(o', sz') | (o', sz') \text{ overlaps } (o, sz)\}, G} \text{(StackWrite)}$$

$$\frac{C \vdash a_1 \approx_\tau a_2 \in G \quad \tau \text{ is not a pointer} \quad \text{sizeof}(\tau) = sz}{C, S, G \xrightarrow{\text{GlobalRead}_{sz}(a_1,v_1),\text{GlobalRead}_{sz}(a_2,v_2)} C \cup \{v_1 = v_2\}, S, G} \text{(GlobalReadVal)}$$

$$\frac{C \vdash a_1 \approx_{\tau*} a_2 \in G}{C, S, G \xrightarrow{\text{GlobalRead}_{sz}(a_1,v_1),\text{GlobalRead}_{sz}(a_2,v_2)} C \cup \text{Ptr}(v_1, v_2), S, G \cup \{v_1 \approx_\tau v_2\}} \text{(GlobalReadPtr)}$$

$$\frac{C \vdash a_1 \approx_\tau a_2 \in G \quad C \vdash v_1 = v_2 \quad \tau \text{ is not a pointer} \quad \text{sizeof}(\tau) = sz}{C, S, G \xrightarrow{\text{GlobalWrite}_{sz}(a_1,v_1),\text{GlobalWrite}_{sz}(a_2,v_2)} C, S, G} \text{(GlobalWriteVal)}$$

$$\frac{C \vdash a_1 \approx_{\tau*} a_2 \in G \quad C \vdash v_1 \approx_\tau v_2 \in G}{C, S, G \xrightarrow{\text{GlobalWrite}_{sz}(a_1,v_1),\text{GlobalWrite}_{sz}(a_2,v_2)} C, S, G} \text{(GlobalWritePtr)}$$

**Figure 3.12:** Inference rules

When formulating these rules, I often preferred simplicity to expressiveness. For example, it would be possible to formulate the rules in a way that allows stack writes to non-matching offsets, or even symbolic offsets, but this would make the implementation more difficult.

The final step after processing the memory traces is to use the final set of SMT constraints to prove the equivalence of the path conditions.

### 3.5.4 Implementation

The simulation tool was implemented as a script inside read-dwarf, which receives a relocatable file and the name of a function to verify. First, it runs the symbolic execution to obtain the symbolic state tree. Then, it verifies a simulation relation between two copies of the same execution tree using the algorithm described in §3.5.3.

We need to distinguish the variables between the two copies of the execution tree. I will use superscript $L$ for "left" and $R$ for "right" execution tree[3]. For example Section$^L$(.bss) and Section$^R$(.bss) are two independent variables representing the address of `.bss` section in two different instances of executing a given file.

An example of this procedure is illustrated in Figure 3.13. We first construct an initial verification context based on the initial state's type information. Then we iterate through the

[3]In code, the variables are wrapped in a union type with `L` and `R` variants

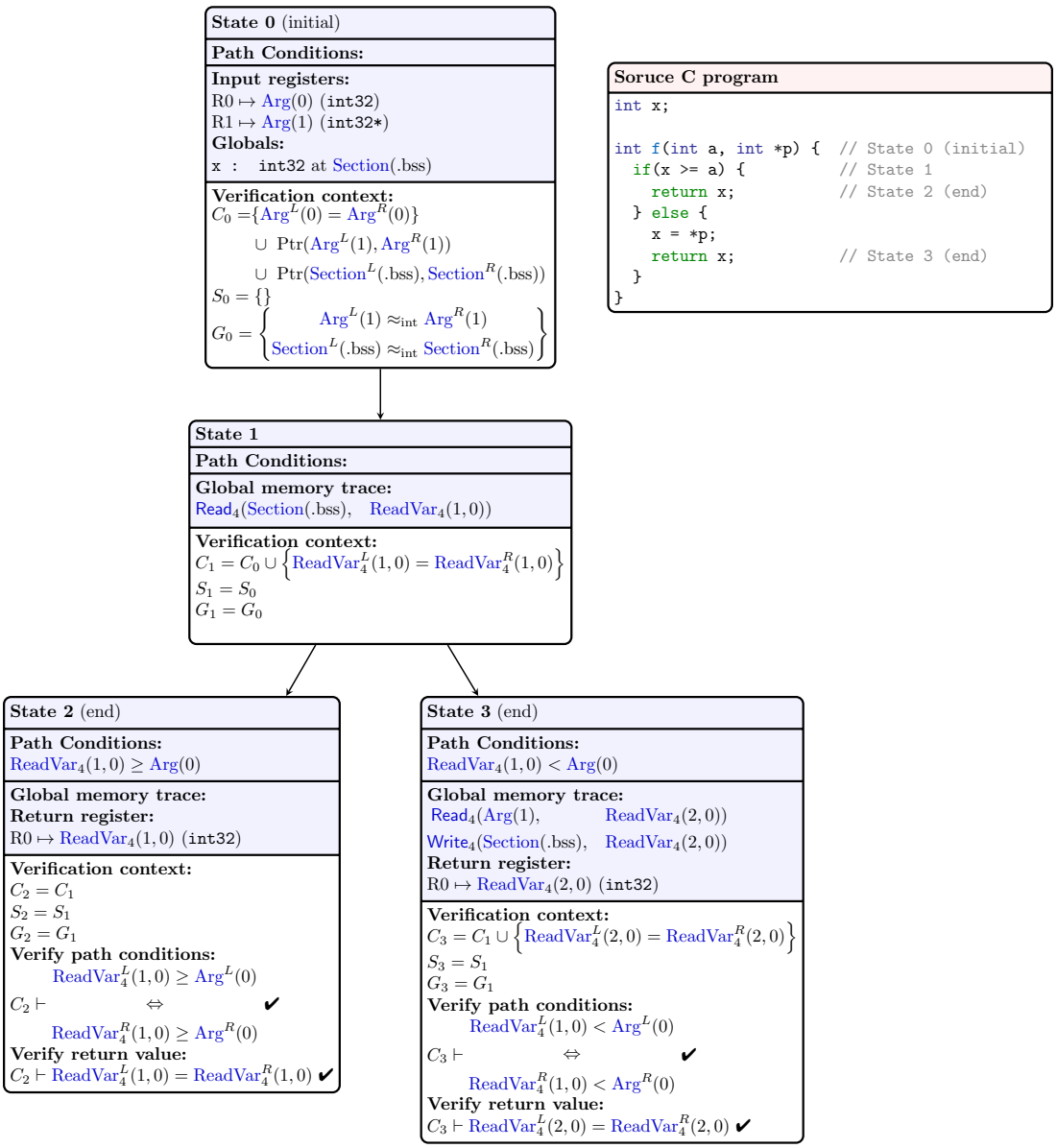



**Figure 3.13:** Simulation verification on the execution tree of an example function. In this example, we can ignore the stack constraints and stack traces, because no arguments are passed on the stack. The states are processed in sequence. **State 0**: The verification context is initialized according to the types of the two arguments and the global variable `x`. **State 1**: This is the state just before the branch. We take the evaluation context from state 0, and update it according to the memory trace (a read from `x`). **State 2**: This corresponds to the if branch. The memory trace is empty, so the evaluation context is the same as in state 1. We verify the path conditions and since it is an end state, we verify the values in the return register. **State 3**: This corresponds to the else branch. We take the evaluation context from state 1, and update it according to the memory trace (a read from `*p` and write to `x`). We verify the path conditions and the return value.

execution tree, computing the verification context for each state. This is done by iterating through the stack and global memory traces and applying the inference rules from Figure 3.12. For each state, we also verify the equivalence between the path conditions in the "left" and "right" tree. Upon reaching a leaf node that represents returning from the function, we also verify that the return values satisfy equivalence constraints expected for the function's return type.

The initial context is constructed using type information about the given function. The type information was previously extracted from DWARF and used to annotate registers and stack memory of the initial state with C types. We can just iterate through the registers and generate constraints based on their types, as described before by the algorithm. We also

generate stack constraints based on the stack type information. To generate constraints about global variables, we iterate through the debug information about all global variables. For each, we use its location description and type to generate a global memory constraint.

If the simulation fails because it is unable to verify the equivalence of path conditions for some state (resp. pair of "left" and "right" state), it reports exactly which path conditions were problematic. It can also fail to verify the equivalence of return values or apply an inference rule while processing a memory operation. It reports what condition caused the failure. My implementation allows false negatives, therefore a negative output has to be checked manually.

## 3.6 Repository overview

The project consists of `linksem`, `read-dwarf` and `isla` repositories, with additional evaluation scripts in the `evaluation` repository. Figures 3.14 and 3.15 provide an overview of the main additions and modifications to `linksem` and `read-dwarf` respectively. Files with only minor modifications are omitted. Multiple parts of `read-dwarf` were changed to "relocatable representation", which mainly involved switching from absolute addresses to section-relative. In `isla`, only `client.rs` was modified per §3.3.3. Figure 3.16 show the overview of `evaluation`.

| | | | |
|---|---|---|---|
| `src` | | +2081 | -1008 |
| `abis` | | | |
| `aarch64` | | | |
| `abi_aarch64_symbolic_relocation.lem` | Relocation specification for AArch64 (§3.1.1) | +182 | |
| `...` | | | |
| `abi_symbolic_relocation.lem` | Utilities for relocation specifications | +77 | |
| `...` | | | |
| `dwarf.lem` | DWARF parsing and analysis. Modified to process relocations and use symbolic types (§3.2). | +1070 | -979 |
| `dwarf_byte_sequence.lem` | Symbolic byte sequence (§3.2.2) | +209 | |
| `elf_symbolic.lem` | The symbolic expressions and the universal relocation representation (§3.1.3). | +150 | |
| `main_elf.lem` | The main program, intended for testing purposes | +16 | -27 |
| `sym.lem` | Lem bindings for `sym_ocaml.ml` | +207 | |
| `sym_ocaml.ml` | OCaml implementation of DWARF symbolic types (§3.2.1) | +160 | |
| `...` | | | |

**Figure 3.14:** Repository overview of linksem

| | | | |
|---|---|---|---|
| `src` | | +2737 | -726 |
| ├── `analyse/` | Html visualization of DWARF. Modified to relocatable representation | +361 | -201 |
| ├── `dw/` | Simplified representation of DWARF. Modified to relocatable representation | +56 | -22 |
| ├── `elf/` | Representation of ELF. Modified per §3.3.1 | +368 | -93 |
| ├── `isla/` | Interaction with Isla-client. Modified per §3.3.3 | +193 | -69 |
| ├── `relsim/` | Verification of semantic relocatability. Modified per §3.5 | +447 | |
| ├── `run` | Top level utilities and scripts for running symbolic execution | +451 | -78 |
| │ ├── `relProg.ml` | Symbolic execution of full relocatable programs (§3.4) | +229 | |
| │ ├── `testRelProg.ml` | Script for testing the symbolic execution (§4.2.1) | +120 | |
| │ └── ... | | | |
| ├── `state/` | State representation. Modified per §3.4.1 (initialization), §3.3.7 (memory operation), including lazy simplification (§3.3.8) | +411 | -63 |
| ├── `trace/` | Representation of the instruction trace. Modified per §3.3.4, §3.3.6 | +174 | -51 |
| ├── `z3/` | Interaction with the Z3 SMT solver. Added a new simplification algorithm (§3.3.8) | +72 | -0 |
| └── ... | | | |

**Figure 3.15:** Repository overview of read-dwarf

| | | |
|---|---|---|
| `/` | | +130 |
| ├── `coverage/` | C programs used as coverage tests | — |
| ├── `simulation/` | C files used for testing the semantic relocatability verfication | — |
| ├── `compile.sh` | Script to compile the test cases | +13 |
| ├── `linksem.sh` | Script to test the extraction of DWARF (§4.1) | +33 |
| ├── `subsitute.py` | Utility used by `linksem.sh` | +30 |
| ├── `symbex.sh` | Script to test symbolic execution (§4.2) | +19 |
| ├── `verify.sh` | Script to run the semantic relocatability verification tool (§4.4) | +16 |
| └── `visualize.sh` | Script to generate html execution logs for the test cases (§4.3) | +19 |

**Figure 3.16:** Repository overview for evaluation

# 4 | Evaluation

The success criterion set out in the project proposal was to enable symbolic execution (with respect to section addresses) of relocatable programs. Additionally, it should allow interpreting their debug information.

The project consists of two core components: the extraction of debug information and the symbolic execution tool. I assessed these separately for their correctness and coverage.

Two tools were made based on the core components: one for visualizing the symbolic execution of a relocatable file, and one for verifying semantic relocatability. These were also assessed for correctness and coverage, as well as usability.

I compiled a set of test programs, some of which were used in multiple parts of the evaluation. The complete set consists of the following:

- 12 short coverage tests exercising the C primitives from Table 2.3.
- A binary of the Android pKVM hypervisor [2], as an example of a larger real-world binary with a complex use of DWARF (only used to test reading debug information).
- A snapshot of the `execute/` directory of the GCC C torture tests from April 2025 [5], consisting of 1676 tests. The tests have a form of short C programs (more details about the structure of the tests in §4.2). I removed programs that rely on GCC-specific features, are specific to a different processor architecture, or import libraries. I ended up with 1426 C programs that I was able to compile for AArch64.
- 20 small example functions to test the semantic relocatability verification tool.

| **Primitive** | **Notes** |
|---|---|
| Control flow primitives | conditionals, loops, function calls |
| Global variables | reading, writing |
| Pointers | including when affected by relocations (e.g. pointing to global variables) |
| Arrays | |
| Structs | |

**Table 2.3:** C primitives to be supported (repeated from page 16)

## 4.1 Reading Dwarf

This part of the evaluation checks whether the modified linksem correctly extracts and interprets the DWARF debug information. The correctness was checked by comparing the output with that of the original linksem on an executable file after linking.

### 4.1.1 Method

The modified linksem was used to extract and print the debug information from each relocatable file. The debug information was extracted using linksem option `--debug-dump=info`. This parses the debug information, runs several analysis functions and pretty-prints their outputs.

To obtain a reference, a linker was run on the file to produce an executable file. The original linksem was then used to extract reference debug information.

To compare the two outputs (example in Figure 4.1), the symbolic expressions in the test output were replaced by concrete values. A simple Python script was used to find all expressions of the form `.section+offset` and replace them by the actual address of the section, incremented by the offset. The section addresses were extracted from the section table in the executable file using `readelf` command. The outputs were further processed before being compared using `diff`.

Firstly, byte sequences in variable location descriptions were replaced by placeholders to avoid dealing with symbolic byte sequences in the output. No information was lost by this transformation because the output also contains the interpretation of the byte sequence as a list of operations.

Secondly, pointers into the string sections were replaced by a placeholder. All strings such as variable names are stored in separate string sections of the ELF file and are referenced by their address. The contents of these sections change during linking. Therefore, the addresses of some strings would not match and needed to be ignored. Again, this does not remove information because the literal string is also shown in the output.

The test was performed on all coverage tests, the 1426 GCC tests and on the pKVM binary.

```
***** .debug_info section - full *******
...
*** compilation unit die tree
...
<1><2a>: Abbrev Number: 2 (DW_TAG_variable)
   <2b> DW_AT_name     : (DW_FORM_strp)
   ↪  AV_sec_offset 0x65 x
   ...
   <35> DW_AT_location : (DW_FORM_exprloc)
   ↪  AV_exprloc 9 With symbolic bytes!
   ↪  [03,00,00,00,00,00,00,00,00]
   ↪  DW_OP_addr .data+0x0
...
***** analysis of location data ********
 x ... .text+0x0 .text+0xc DW_OP_addr
  ↪  .data+0x0 ...
...
```

**(a)** Relocatable file

```
***** .debug_info section - full *******
...
*** compilation unit die tree
...
<1><2a>: Abbrev Number: 2 (DW_TAG_variable)
   <2b> DW_AT_name     : (DW_FORM_strp)
   ↪  AV_sec_offset 0x65 x
   ...
   <35> DW_AT_location : (DW_FORM_exprloc)
   ↪  AV_exprloc 9
   ↪  [03,e8,0f,41,00,00,00,00,00]
   ↪  DW_OP_addr 0x410fe8
...
***** analysis of location data ********
 x ... 0x4000e8 0x400104 DW_OP_addr
  ↪  0x410fe8 ...
...
```

**(b)** Executable file

**Figure 4.1:** Linksem debug info dump outputs for relocatable and executable file. Red marks the section addresses that are substituted in the output. The orange parts are ignored (replaced by a placeholder).

### 4.1.2 Results

All tests except one GCC test succeeded. The failing test (Figure 4.2) was investigated. Investigation revealed the failure was due to a transformation done by the linker, rather than a problem in my implementation.

```
typedef unsigned char uint8_t;
uint8_t foo[1][0];
extern void abort (void);
int main()
{
  if (sizeof (foo) != 0)
    abort ();
  return 0;
}
```

**Figure 4.2:** Failing linksem test (pr42570.c)

The `.bss` section, which contains only the zero length array `foo`, is removed by the linker. The debug information in the relocatable file describes `foo` in a location `.bss+0x0`, but because `.bss` was removed, we are unable to substitute it by a concrete address. Interestingly, the DWARF in the executable still contains an entry describing `foo` being located at an address that is not part of any section. This could be considered a bug in the linker.

#### Performance

Although performance is not a priority, it is important that execution time remains within reasonable bounds. I compared the performance of the modified linksem processing relocatable files to the original linksem processing executable files after linking. On most tests, the processing of a relocatable file was around 1.4 times slower compared to the original linksem processing an executable. Only a few were more than two times slower. On pKVM, as an example of a larger real-world program, the processing of a relocatable file was 2 times slower (31 seconds and 62 seconds). The performance remains sufficient for the intended use, despite the decrease.

## 4.2 Symbolic execution

### 4.2.1 Method

The testing framework was designed around the GCC tests, and the other tests were adapted to a compatible format. The tests have a form of C programs that should exit with status 0 if correctly executed. When we execute them symbolically (with respect to their section addresses), we expect to obtain a state tree where all execution paths are either marked as impossible[1] or exit with status 0.

The programs exit either by returning from `main` or by calling standard library functions `exit` or `abort`. To emulate calling these library functions, I set up the symbolic execution to terminate when the program counter reaches an address of either of them.

After the full symbolic state tree is constructed, we check if all terminating states, not marked as impossible, correspond to exiting with status 0. If the symbolic execution terminated by exiting from `main`, the return register is checked to determine the status code. If it terminated by calling `exit`, the register containing the function argument is checked.

Note that the test succeeds only if all failing paths are explicitly marked as impossible. Even if the state tree produced by read-dwarf is valid but contains impossible paths that have not been identified, it is reported as failed.

[1] read-dwarf determined that the path conditions are unsatisfiable

### 4.2.2 Results

#### Coverage Tests

All coverage tests succeeded.

#### GCC Tests

Out of the 1426 compiling C programs, I further removed the following:

- 236 tests containing special directives. The directives specify extra compilation options or requirements. For simplicity, these were all skipped.
- 145 tests calling library functions other than `exit` and `abort`.
- 112 tests using types not supported by read-dwarf (floating point types, bit-fields, unnamed structs).

One test was discovered not to be semantically relocatable (shown in the Introduction in Figure 1.1), i.e. it can fail if the sections are assigned some specific addresses. Of the remaining 932 tests, 16 did not finish within a set time limit (10 minutes), and 58 failed due to limitations of read-dwarf:

- 21 failed because of escaping. Escaping happens when a pointer to a local variable is stored in a global variable. This is not supported by the read-dwarf's provenance tracking[2].
- 37 failed because read-dwarf was unable to simplify memory reads.

**Symbolic reads** Programs sometimes use a 64 bit memory read for data that is smaller than 64 bits. Since the read spans into uninitialized memory, read-dwarf represents the read value using a fresh read variable. Even if only the part of the read that contains initialized data is used, there is currently no mechanism in read-dwarf that can simplify the expressions once a read variable is introduced.

## 4.3 Execution visualization

The project also included a feature to visualize the symbolic execution of a program. This allows one to manually inspect the execution and see how values of individual variables depend on section addresses. It prints the instructions being executed, interleaved with debug information. The tool was tested on the 12 coverage tests. The output was manually inspected for:

- correctness — shows the correct execution path and accurate values of debug variables,
- completeness — displays the values of all live variables,
- interpretability — it is easy to follow the execution of the program and read the values of variables.

The tool displayed correct and complete outputs on these tests. An example is shown on Figure 4.3. An experienced user can easily find relevant information in the output, but it would likely cause problems to an unfamiliar user.

[2] All global pointers are assumed to have `Main` provenance, meaning they cannot point to places on the stack

```c
typedef struct { int x; int y; } A;
A a;

int main() {
    A *c = &a;
    A b = { .x=42, .y=47 };
    b.x += b.y;
    *c = b;
}
```

**(a)** Source code

```
  main:12.10 (struct.c) sbepe      *c = b;
~ .text+00000034:  f94003e8  ldr              x8, [sp]
a = typedef.A{ x = 0; y = 0 }
c = |section:.bss|
b = typedef.A{ x = 89; y = 47 }
~ .text+00000038:  f9000128  str              x8, [x9]
a = typedef.A{ x = 89; y = 47 }
c = |section:.bss|
b = typedef.A{ x = 89; y = 47 }
~ .text+0000003c:  2a1f03e0  mov              w0, wzr
a = typedef.A{ x = 89; y = 47 }
c = |section:.bss|
b = typedef.A{ x = 89; y = 47 }
```

**(b)** Fragment of the output

**Figure 4.3:** A fragment of the rendered HTML output from the visualization tool

## 4.4 Simulation

The simulation tool verifies semantic relocatability of simple C functions. It either confirms that the function is semantically relocatable or provides evidence of the opposite. The tool allows false negatives; therefore, the evidence has to be manually checked.

I prepared 20 small example functions to test the tool on, consisting of positive (semantically relocatable) and negative (not semantically relocatable) examples. Unfortunately, the current limitations of the tool prevent it from being meaningfully used on larger real-world programs.

I checked if the tool gives the correct result. In case of a negative result, I assess the interpretability of the result and how easy it is to confirm or refute the result.

### 4.4.1 Coverage

Apart from the limitations inherited from read-dwarf, the tool is mostly limited by the complexity of the types and pointer operations used in a given program. It supports different sized integer types and pointers to them. It does not support structs and arrays.

Since the simulation is based on the construction of a full symbolic execution tree, it is restricted to programs with a finite execution tree, small enough to be efficiently constructed. This rules out most programs containing loops.

### 4.4.2 Results

The tool successfully verified the semantic relocatability of functions that contain branching, read and write through pointers and global variables, perform null pointer checks, compare pointers within the same section, and perform integer arithmetic.

It produced false negatives on functions that compare pointers for equality, temporarily write values into global memory that are not equivalent based on their type (even when they are later overwritten), or rely on specific values written into global memory. An example is shown in Figure 4.5.

It successfully identified a violation of semantic relocatability in all negative examples. Examples are shown in Figures 4.4 and 4.6. The outputted expressions can be in a form that is difficult to interpret. If they contain read variables, it may be necessary to consult the execution tree.

See Appendix B for the results of all test cases.

```c
int ptr_val(char *x) {
    return (int)x;
}

char f(char *x) {
    int y = ptr_val(x);
    if(y == 0)
        return 0;
    else
        return *x;
}
```

**(a)** Source code

```
States not equivalent on path conditions
  L: [ !(|L:arg:0|[0:31] = 0x0:32) ]
  R: [ !(|R:arg:0|[0:31] = 0x0:32) ]
```

**(b)** Output

**Figure 4.4:** True negative result. Interpretation: the program behaviour could diverge depending on whether the bottom 32 bits of the function argument (a pointer) are zero. This can indeed be affected by the memory addresses assigned for sections.

```c
int f(int *a, int *b) {
    if (a == b)
        return *a;
    else
        return 0;
}
```

**(a)** Source code

```
States not equivalent on path conditions
  L: [ !(|L:arg:0| = |L:arg:1|) ]
  R: [ !(|R:arg:0| = |R:arg:1|) ]
```

**(b)** Output

**Figure 4.5:** False negative result. Interpretation: the tool reports the program behaviour could diverge depending on whether the two pointer arguments are equal. In reality, this is not dependent on specific section addresses.

```c
int x;

int f(int *y) {
    if (y < &x)
        return 0;
    else
        return x;
}
```

**(a)** Source code

```
States not equivalent on path conditions
  L: [ !(0x0.(0x1 + |L:arg:0| + ~|L:section:.bss|) = 0x1:128 +
  ↪  0x0.|L:arg:0| + 0x0.(~|L:section:.bss|)) ]
  R: [ !(0x0.(0x1 + |R:arg:0| + ~|R:section:.bss|) = 0x1:128 +
  ↪  0x0.|R:arg:0| + 0x0.(~|R:section:.bss|)) ]
```

**(b)** Output

**Figure 4.6:** True negative result (difficult to interpret). The condition could be simplified to `|arg:0|<|section:.bss|`

.

# 5 | Conclusions

The project achieved all core goals and a substantial part of the extensions. It enabled symbolic execution of relocatable files with respect to the addresses of their sections, and evaluation of the debug information, meeting all the success criteria.

Substantial progress was made in developing an automatic verification tool for semantic relocatability (proposed as an extension of the project). It was successfully applied to simple programs. As expected, the first version has limitations that need to be addressed before it could be applied in realistic situations. Some of the limitations are implementation related (e.g. support for composite types) and can be easily addressed. Others are more fundamental and open-ended (e.g. handling loops).

## 5.1 Reflection

A large part of the project involved the modification of existing tools. Considerable time was devoted to understanding their implementation, the design decisions made, and working around them to achieve the project goals, leaving less for extensions. Getting familiar with the tools before the project would have been beneficial for both planning and implementation.

## 5.2 Future work

**Core functionality** The modifications I made to read-dwarf broke some previous functionality, including the ability to process executable files. Work is required to unify my version with the original version before it can be upstreamed to the read-dwarf repository.

The implementation can be extended to cover more relocation types. Currently, there is no support for dynamic linking. It would require handling dynamic relocations applied to the Global Offset Table[1] and static relocations that refer to entries of the table. Support can also be extended to other architectures beyond AArch64. A good candidate is RISC-V, since it is already supported by (the original) read-dwarf.

**Verification tool** It remains to add support for composite types. A larger challenge is to support unbounded loops. This will require a different approach, where the simulation relation is constructed between points of the program (such as loop entry points), rather than the states of a complete execution tree. Symbolic execution will be run only between the points to verify the simulation. Automatically inferring the simulation relation for arbitrary programs is impossible, and how to address practical cases remains an open question. The type information from DWARF can be leveraged, as currently used in function precondition generation.

The simulation tool can be adapted to verify linkers. Linkers are allowed to perform simple optimizations when producing an executable from relocatable files. These could be verified using a simulation between the original relocatable file and the resulting executable.

[1] A table of addresses that is filled by a dynamic linker when loading a shared library

# Bibliography


[1] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell. *Isla: Symbolic execution tool for Sail ISA specifications*. https://github.com/rems-project/isla. 2020.

[2] Android. *pKVM Hypervisor Source Code – Tag: pkvm-core-6.4*. 2023. URL: https://android-kvm.googlesource.com/linux/+/refs/tags/pkvm-core-6.4.

[3] ARM Ltd. *ELF for the Arm® 64-bit Architecture (AArch64)*. URL: https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst.

[4] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format - Version 4*. URL: https://dwarfstd.org/dwarf4std.html.

[5] FSF. *GNU Compiler Collection, C Torture Test Suite*. Accessed: 2025-04. Commit: 19ba913. URL: https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite/gcc.c-torture/execute.

[6] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[7] S. Kell, D. Mulligan, and P. Sewell. *Linksem: Semantic model for aspects of ELF static linking and DWARF debug information*. https://github.com/rems-project/linksem. 2014.

[8] T. Pérami, D. C. Makwana, N. Krishnaswami, and P. Sewell. *read-dwarf: Binary analysis tool*. https://github.com/rems-project/read-dwarf. In collab. with T. Pérami, S. Ser, S. Flur, R. M. Norton, R. Kumar, J. French, B. Campbell, and T. Bauereiss. 2019.

# A | Supported relocation types

The ABI specification for AArch64 defines over 100 relocation types [3], but most programs use only a small fraction of them. Table A.1 lists the relocations that my implementation currently supports, along with their representation as described in §3.1.1. These were the only relocation types encountered across all tested programs. These relocation types fall into four groups:

- Miscellaneous — used to mark dependencies to prevent a static linker from removing a section, but have no direct effect.
- Data — applied to data sections.
- PC-relative addressing — applied to instructions that generate PC-relative addresses (including load/store instructions).
- Control flow — applied to control flow instructions.

| **Name** | **Representation** | | | |
|---|---|---|---|---|
| | **Operation** | **Safety Checks** | **Mask** | **Target** |
| **Miscellaneous relocations** | | | | |
| R_AARCH64_NONE | None (no effect) | | | |
| withdrawn | None (no effect) | | | |
| **Data relocations** | | | | |
| R_AARCH64_ABS64 | S+A | | (63, 0) | Data64 |
| R_AARCH64_ABS32 | S+A | Overflow($-2^{31}, 2^{32}$) | (31, 0) | Data32 |
| R_AARCH64_PREL64 | S+A-P | | (63, 0) | Data64 |
| R_AARCH64_PREL32 | S+A-P | Overflow($-2^{31}, 2^{32}$) | (31, 0) | Data32 |
| **Relocations to generate PC-relative addresses** | | | | |
| R_AARCH64_ADR_PREL_PG_HI21 | Page(S+A)-Page(P) | Overflow($-2^{32}, 2^{32}$) | (32, 12) | ADRP |
| R_AARCH64_ADD_ABS_LO12_NC | S+A | | (11, 0) | ADD |
| R_AARCH64_LDST8_ABS_LO12_NC | S+A | | (11, 0) | LDST 0 |
| R_AARCH64_LDST16_ABS_LO12_NC | S+A | Alignment(1) | (11, 1) | LDST 1 |
| R_AARCH64_LDST32_ABS_LO12_NC | S+A | Alignment(2) | (11, 2) | LDST 2 |
| R_AARCH64_LDST64_ABS_LO12_NC | S+A | Alignment(3) | (11, 3) | LDST 3 |
| R_AARCH64_LDST128_ABS_LO12_NC | S+A | Alignment(4) | (11, 4) | LDST 4 |
| **Control flow relocations** | | | | |
| R_AARCH64_CALL26 | S+A-P | Overflow($-2^{27}, 2^{27}$), Alignment(2) | (27, 2) | CALL |
| R_AARCH64_CONDBR19 | S+A-P | Overflow($-2^{20}, 2^{20}$), Alignment(2) | (20, 2) | CONDBR |
| R_AARCH64_JUMP26 | S+A-P | Overflow($-2^{27}, 2^{27}$), Alignment(2) | (27, 2) | B |

**Table A.1:** List of supported AArch64 relocation types and their corresponding representation.

The support can be easily extended to other relocation types within these groups. Other groups that are currently **not** supported include:

- Group relocations — applied to a sequence of `mov` instructions that construct the relocated value. Most of them can be easily added, except a few whose behaviour depends on the sign of the relocated value.
- Relocations used to support dynamic linking:
  - Static relocations that compute indices into the Global Offset Table (GOT).
  - Dynamic relocations (processed by the dynamic linker) that fill the GOT with run-time addresses.
- Relocations used to support thread-local storage

# B | Results of Semantic Relocatability Tests

## B.1 True Positives

```
long long x = 1;

void f(long long a) {
    x = a;
}
```

```
Simulation successful
```

**Figure B.1:** positive/example.c

```
int x;

int f(int *y, int a) {
    return (a + *y * 7) % x;
}
```

```
Simulation successful
```

**Figure B.2:** positive/integer_arithmetic.c

```
int x;

int f(int *y, int a) {
    a += *y;
    if(x < a)
        x = a;
    return a;
}
```

```
Simulation successful
```

**Figure B.3:** positive/integer_comparison.c

```
int f(int *x) {
    if(x == 0)
        return 0;
    else
        return *x;
}
```

```
Simulation successful
```

**Figure B.4:** positive/null_check.c

```
long long ptr_val(int *x) {
    return (long long) x;
}

int f(int *x) {
    long long y = ptr_val(x);
    if(y == 0)
        return 0;
    else
        return *x;
}
```

```
Simulation successful
```

**Figure B.5:** positive/null_check_after_cast.c

```
void f(int *x, int *y) {
    *y = *x;
}
```

```
Simulation successful
```

**Figure B.6:** positive/pointer_read_write.c

```
int x, y;

int f() {
    if (&x < &y)
        return 0;
    else
        return x;
}
```

```
Simulation successful
```

**Figure B.7:** positive/pointer_safe_comparison.c

```
int f(int x, int y) {
    if (&x < &y)
        return 0;
    else
        return x;
}
```

```
Simulation successful
```

**Figure B.8:** positive/pointer_safe_comparison2.c

```
void f(int *x) {
    *x = 42;
}
```

```
Simulation successful
```

**Figure B.9:** positive/pointer_write.c

```
int y;

void f(int *x) {
    *x = y;
}
```

```
Simulation successful
```

**Figure B.10:** positive/pointer_write2.c

```
void f(int *x, int y) {
    *x = y;
}
```

```
Simulation successful
```

**Figure B.11:** positive/pointer_write3.c

```
void f(int **x, int *y) {
    *x = y;
}
```

```
Simulation successful
```

**Figure B.12:** positive/pointer_write4.c

## B.2 True Negatives

```c
int ptr_val(char *x) {
    return (int)x;
}

char f(char *x) {
    int y = ptr_val(x);
    if(y == 0)
        return 0;
    else
        return *x;
}
```

```
Simulation failed:

States not equivalent on path
↪  conditions
  L: [ !(|L:arg:0|[0:31] = 0x0:32)
   ↪  ]
  R: [ !(|R:arg:0|[0:31] = 0x0:32)
   ↪  ]
```

**Figure B.13:** negative/null_check_after_lossy_cast.c

```c
int ptr_val(int *x) {
    int *p = 0;
    return (int) (x-p);
}

int f(int *x) {
    int y = ptr_val(x);
    if(y == 0)
        return 0;
    else
        return *x;
}
```

```
Simulation failed:

States not equivalent on path
↪  conditions
  L: [ !(|L:arg:0|[2:33] = 0x0:32)
   ↪  ]
  R: [ !(|R:arg:0|[2:33] = 0x0:32)
   ↪  ]
```

**Figure B.14:** negative/null_check_after_lossy_cast2.c

```
int x;

int f(int *y) {
    if (y < &x)
        return 0;
    else
        return x;
}
```

```
Simulation failed:

States not equivalent on path
↪  conditions
  L: [ !(0x0.(0x1 + |L:arg:0| +
   ↪  ~|L:section:.bss|) = 0x1:128
   ↪  + 0x0.|L:arg:0| +
   ↪  0x0.(~|L:section:.bss|)) ]
  R: [ !(0x0.(0x1 + |R:arg:0| +
   ↪  ~|R:section:.bss|) = 0x1:128
   ↪  + 0x0.|R:arg:0| +
   ↪  0x0.(~|R:section:.bss|)) ]
```

**Figure B.15:** negative/pointer_unsafe_comparison.c

```
int x;

int f(int y) {
    if (&y < &x)
        return 0;
    else
        return x;
}
```

```
Simulation failed:

States not equivalent on path
↪  conditions
  L: [
    !(0x0.(-0x7 + |L:reg:1:SP_EL2|
     ↪  + ~|L:section:.bss|)
        = 0x1:128 + 0x0.(-0x8 +
         ↪  |L:reg:1:SP_EL2|) +
         ↪  0x0.(~|L:section:.bss|))
  ]
  R: [
    !(0x0.(-0x7 + |R:reg:1:SP_EL2|
     ↪  + ~|R:section:.bss|)
        = 0x1:128 + 0x0.(-0x8 +
         ↪  |R:reg:1:SP_EL2|) +
         ↪  0x0.(~|R:section:.bss|))
  ]
```

**Figure B.16:** negative/pointer_unsafe_comparison2.c

```
long long x;

void f(int *a) {
    x = (long long)a;
}
```

```
Simulation failed:

Unable to verify Eq between  L:
↪  |L:arg:0| R: |R:arg:0|
```

**Figure B.17:** negative/pointer_to_integer_cast.c

## B.3 False Negatives

```
int f(int *a, int *b) {
    if (a == b)
        return *a;
    else
        return 0;
}
```

---

```
Simulation failed:

States not equivalent on path
↪  conditions  L: [ !(|L:arg:0| =
↪  |L:arg:1|) ] R: [ !(|R:arg:0| =
↪  |R:arg:1|) ]
```

**Figure B.18:** positive-failed/aliasing.c

```
int f(int *a, int *b) {
    return a == b;
}
```

---

```
Simulation failed:

Return values not equivalent
Condition: Eq between
  L: 0x0:32.(0x1:32 + (if |L:arg:0|
  ↪  = |L:arg:1| then 0x0:32 else
  ↪  -0x1:32))
  R: 0x0:32.(0x1:32 + (if |R:arg:0|
  ↪  = |R:arg:1| then 0x0:32 else
  ↪  -0x1:32))
```

**Figure B.19:** positive-failed/aliasing2.c

```
long long x;

int is_x_zero() {
    return x == 0;
}

int f(int *y) {
    x = (long long)y;
    if (is_x_zero()) {
        return 0;
    } else {
        return x = 1;
    }
}
```

---

```
Simulation failed:

Unable to verify Eq between  L:
↪  |L:arg:0| R: |R:arg:0|
```

**Figure B.20:** positive-failed/global_temp_write.c

# C | Read-dwarf debug information analysis

Read-dwarf contains a module that processes the DWARF output of linksem and displays it in a more human-readable format using HTML. My contributions enabled applying the tool to relocatable files, where the addresses are displayed in the `.section+offset` form. The output for pKVM is shown in Figure C.1.

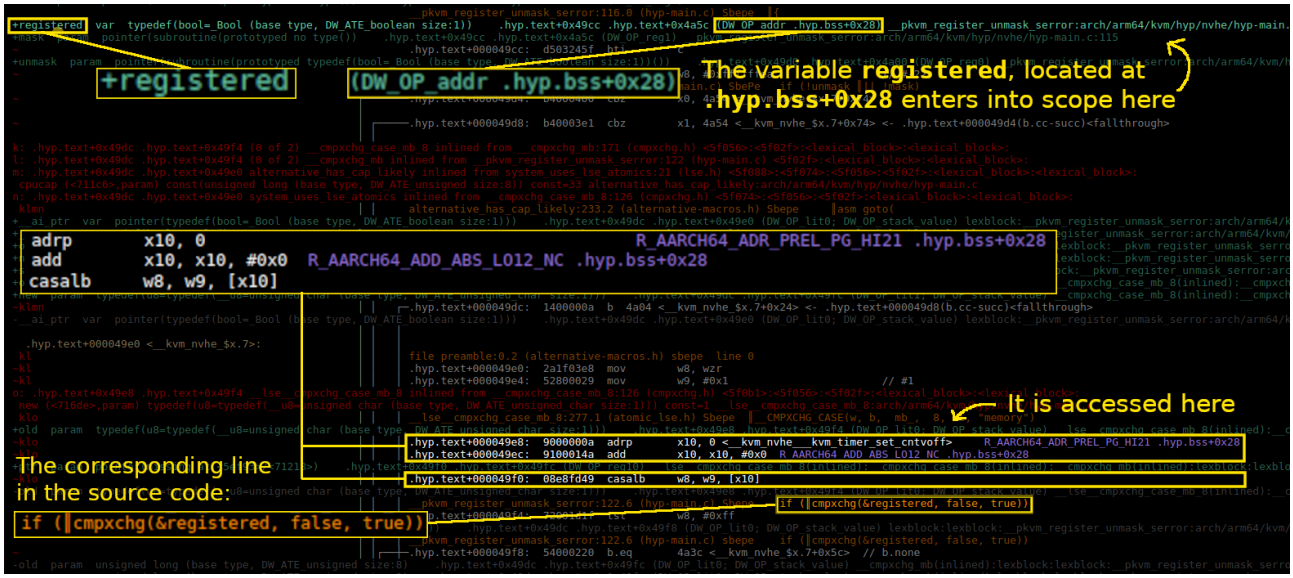



**Figure C.1:** Fragment of the evaluated debug information from pKVM, displayed using HTML. The code fragment is accessing a static variable `registered`, located in the `.bss` section. The output displays the program instructions (white), interleaved with parts of the debugging information. The descriptions of variables when they enter the scope are green, the corresponding lines in the source code are orange, relocations are purple.

# Project Proposal

# Formal Verification of ELF Relocations

## Introduction and description

Given the complexity of modern optimizing compilers, bugs can occur that cause miscompilation. This can be a major problem for critical software. To guarantee the correctness of a compiled program, one can use a formally verified compiler, but these often lack in features. An alternative approach is Translation Validation.

Translation Validation is a technique that aims to automatically prove for a given program that it was compiled correctly, treating the compiler as a black box. It is a big and complex task with ongoing research. In my project, I will tackle a small subproblem of verifying relocations.

A program compiled into an ELF file consists of sections that can be placed at different addresses in memory. It has a relocation table describing how to modify adresses within the file based on where each section ends up. My project aims to develop a tool that validates that the behaviour of a program is independent of where each section is placed.

Debug information can be used to find a correspondence between the compiled program and its source. It is specified inside the ELF file in a DWARF format. It contains expressions that need to be evaluated to obtain the debug information. The information includes a mapping of the machine code to lines in the source code and descriptions of where each variable is located at each point of the program's execution. I will use it to assist with the verification.

At minimum, my project should be able to verify simple programs taking no input, through symbolic execution. It should also be able to evaluate the debug information at any point of the symbolic execution.

Extensions will cover more complex cases, aiming to verify individual functions executing in unknown contexts.

## Starting point

I will be using an existing executable specification of DWARF, that was developed in Cambridge as part of Linksem (formalisation of ELF linking and DWARF debug information). It covers a substantial part of the DWARF 4 standard. I will need to modify it to allow symbolic evaluation with relocations.

I will be using Isla for the symbolic evaluation of Arm instructions. I have not used it before and will need to familiarise myself with it and its capabilities.

## Work to be done

I will need to extend Linksem to extract the information I need from the ELF file. First are the relocations, that I want to extract as expressions that can

be symbolically evaluated and applied. I also need to be able to symbolically evaluate the debug information with symbolic relocations applied. Based on my observations, the relocations only appear in simple DWARF expressions, which makes the symbolic evaluation simple.

The relocations need to be processed to construct a symbolic state of memory at the start of the program's execution. The memory address where each section starts will be specified by a symbolic variable and the relocations will be applied. I will give this state to Isla and let it execute the program. The verification succeeds if the output value is independent of the symbolic variables. To evaluate debug information at a given point, I will run Isla up to that point and evaluate the DWARF expressions on the current state.

## Extensions

Extensions will involve verifying individual functions with inputs and/or side effects. For that, I plan to utilize Bisimulation. Bisimulation is a relation between states of two programs that behave in the same way. Let's consider two different runs of the program, differing in the placement of their sections. If we construct a Bisimulation between them, we can prove that they are equivalent.

To do that, I will find a correspondence between the states of the memory and registers at each program point. I will use the debug information to find all parts of memory reachable in the current scope, and determine how their values should relate between the two runs. Using Isla, I can verify that each instruction respects this relation, finishing the proof.

I will split this into multiple subtasks:

1. Constructing a Bisimulation for a single instruction. Alternatively, constructing it for a short sequence of instructions, treating them as a single state transition
2. Putting together the single instruction steps to verify a simple branch-free function
3. Handling loads/stores that use two instructions, the first of which only loads upper address bits into a register
4. Handling branching
5. Handling function calls
6. Handling compiler optimizations

## Evaluation

I will prepare a selection of programs demonstrating the usage of all C primitives described in the "Success Criterion". My tool will be run on each to produce a result of symbolically executing the program, which will be checked against an expected result determined by hand.

Example programs will also be constructed to evaluate each extension.

## Success Criterion

Support symbolic execution of self-contained, relocatable programs, with no input or side-effects, written in C and compiled for aarch64. It must assume that the layout of the program's sections in memory can be arbitrary, and represent it symbolically.

It has to support all of the following C primitives:

- Control flow (conditionals, loops, function calls)
- Global variables (reads and writes)
- Pointers (including cases where their value is affected by relocations - e.g. pointing to global variables)
- Arrays
- Structs

Allow symbolically evaluating debug information at any point of the symbolic execution.

## Work plan

### Michaelmas

- **Week 1-2**
  - Work on the project proposal
  - Get familiar with Linksem
  - Get familiar with Isla
  - Come up with programs to test the project on

  Milestones: Project proposal submitted

- **Week 3-4**
  - Extend Linksem to process relocations into evaluable expressions
  - Allow propagating the relocations to the DWARF expressions and evaluating them given a concrete layout of the sections in memory

  Milestones: Linksem producing relocations as expressions, and the debug info after applying relocations

- **Week 5-6**
  - Modify Linksem to allow evaluating DWARF with symbolic values. Apply symbolic relocations to DWARF before evaluating
  - Learn to use Isla and find out how to provide the symbolic state of memory at the start of the program

  Milestones: Linksem producing symbolic debug info

- **Week 7-8**

  - Assuming a symbolic address for each section, construct a symbolic state of memory, applying the relocations outputted by Linksem
  - Let Isla execute on this symbolic state and check the output

  Milestones: A complete pipeline capable of verifying programs with no input

**Christmas break**

- Allow evaluating debug information at any point of the symbolic execution
- Research and prepare a plan to construct a Bisimulation

Milestones: A fully functional tool that can symbolically execute a relocatable file and evaluate debug information

**Lent term**

- **Week 1-2**
  - Writing a progress report and preparing the presentation
  - Start working on Bisimulation, handling a single instruction

  Milestones: Progress report submitted and presentation ready
- **Week 3-4**
  - Construct a Bisimulation for a whole function
  - If time permits, handle multi-instruction loads/stores
- **Week 5-6**
  - Continue work on Bisimulation, covering more complex cases
- **Week 7-8**
  - Start writing dissertation

  Milestones: Written Introduction and Peraration sections

**Easter break**

- Continue writing dissertation
- Gather feedback and make corrections to the dissertation

Milestones: Dissertation close to finished, with all sections written, leaving time for final corrections

**Easter Term**

- **Week 1-2**
  - Final corrections to the dissertation

  Milestones: Dissertation and source code submitted

## Resource declaration

I will use my personal laptop (Lenovo Yoga 6, AMD Ryzen 5 7530u, 16GB RAM, 512GB SSD, Ubuntu 22.04) for the project. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. In case of failure, I will use my second laptop. All my work will be backed on GitHub.