Summary

Embedded systems for use in space and high altitude airspace have had to deal with the danger of cosmic rays generating bitflips in their hardware for decades. Traditionally, the problem has been alleviated using several redundant pieces of hardware, such that any single bitflip can be detected. As the microprocessors in consumer electronics become ever smaller the risk that any of them become subject to the same type of error increases. Research has gone into novel ways of preventing bitflips from causing problems on a software only level, as this allows for much cheaper solutions. In this report we take a different approach and develop a novel tool which can be used to analyse if a given assembly program is vulnerable to a single bit flipping in a general purpose register or not. The analysis is defined as a data flow analysis using a monotone framework. Background information on both subjects is presented in the report, such that we have a solid foundation on which to build our analyses. The assembly language we implement the tool for, TinyARM, is based on ARM and is a language which we have given a formal definition of in previous work. We discuss how to represent parsed TinyARM programs as control flow graphs that can be traversed by the analysis and are structured such that they support path sensitive analyses. Initially, we define and implement a version of an interval analysis that can track the state of control flags. We define and eventually implement an extension of the interval analysis, which at each instruction adds an additional set of possible states to the analysis result, such that it models a bit flipping just before the execution of that instruction. We discover efficient ways of simulating bitflips and the status of control flags without resorting to bruteforcing, which makes the analysis feasible to run on architectures of arbitrary bit width. The tool we create also includes an implementation of a TinyARM interpreter and can generate output in different formats. Finally, we show how the analysis handles programs protected by some of the software only schemes proposed in earlier research and discuss how our analysis could be improved and extended with support for broader fault models.

Data Flow Analysis for Discovering Bitflip Vulnerabilities in TinyARM

Master's Thesis

Group:
DS107F20

Supervisor:
René Rydhof Hansen

June 4, 2020

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
https://www.cs.aau.dk

Title:
Data Flow Analysis for Discovering Bitflip Vulnerabilities in TinyARM

Theme:
Master's Thesis

Project Period:
Spring 2020

Group:
DS107F20

Participants:
Anton Christensen
Henrik H. Sørensen

Supervisor:
René Rydhof Hansen

Pages:
35

Date of Completion:
June 4, 2020

Number of Copies:
1

Abstract:

In this report we present two data flow analyses which operate on the assembly language TinyARM that is heavily inspired by ARM. Based on the formal definition of the language from our previous work, we create a path sensitive interval analysis which takes into account the possible states of control flags during a program's execution. We build on the interval analysis to create a novel analysis which is also capable of simulating how the intervals behave in the event that a single bit flips during execution, following a formally defined fault model from our previous work. Both analyses are defined through a monotone framework, however we are unable to prove that they are sound, given that we only argue for the monotonicity of some transfer functions involved in them. We test our bitflip analysis on relevant code examples to show which problems it can detect and which it cannot, while also presenting additional features of our analysis tool. Finally, we discuss several improvements that could be made to the analysis to increase both its precision and usefulness.

Anton Christensen

Anton Christensen
achri15@student.aau.dk

Henrik H. Sørensen
hsaren14@student.aau.dk

Contents

1 Introduction

As our previous work shows, methods for protecting programs against Single Event Upsets (SEUs) has been the focus of past research [1]. However, it has become apparent that using well known data flow analysis techniques to reason about the safety of code in the presence of an SEU is subject of less research.

In this report we introduce a novel data flow analysis that operates at the assembly level and extends an interval analysis to help determine if a piece of code is vulnerable to SEUs in general purpose registers. The code in Listing 1 shows a slightly simplified piece of real world code that is included in modern versions of Linux and used for authentication through the utility Pluggable Authentication Module (PAM) [2]. The code is used to perform several checks to make sure that a user is authenticated in a secure manner, including asking for a password and comparing it to a stored one. However, all of these checks are entirely skipped if the check on line 2 fails. This is so that the root user is never asked for a password and really should not be a cause for concern as the user ID is only 0 for the root user.

```
 1 int auth_pam(const char *service_name, unsigned int uid, const char *username) {
 2     if (uid != 0) {
 3         void *pamh = NULL;
 4         void *conv = NULL;
 5         int retcode;
 6 
 7         retcode = pam_start(service_name, username, &conv, &pamh);
 8         if (pam_fail_check(pamh, retcode))
 9             return FALSE;
10 
11         retcode = pam_authenticate(pamh, 0);
12         if (pam_fail_check(pamh, retcode))
13             return FALSE;
14 
15         retcode = pam_acct_mgmt(pamh, 0);
16         if (retcode == PAM_NEW_AUTHTOK_REQD)
17             retcode =
18                 pam_chauthtok(pamh, PAM_CHANGE_EXPIRED_AUTHTOK);
19         if (pam_fail_check(pamh, retcode))
20             return FALSE;
21 
22         retcode = pam_setcred(pamh, 0);
23         if (pam_fail_check(pamh, retcode))
24             return FALSE;
25 
26         pam_end(pamh, 0);
27         /* no need to establish a session; this isn't a
28          * session-oriented activity...  */
29     }
30     return TRUE;
31 }
```

Listing 1: Simplified code used centrally in PAM authentication

But consider what happens if a bitflip occurs in the register that stores the `uid` when this code is run. As only a single bit is likely to flip, the value before the bitflip must be some power of two for the value to flip to 0. It is a common convention for new users to be given `uid`s starting at 1000, which means that by creating 25 users one can obtain a user with `uid` = 1024 in which case a single bitflip in the right register at the right time can bypass the entire authentication system. Obviously this is unlikely to happen by mistake, but the idea that some bad actor could take advantage of this is not unthinkable. The analysis we present in this report shows that it is possible to create a static program analysis that can be used to identify this type of security flaw.

The report is structured as follows: in Section 2 we discuss our motivation for creating an analysis for detecting bitflips and give a cursory overview of what such an analysis should be capable of. We then cover some background theory and give an overview of lattices, data flow analyses, monotone frameworks and how the three complement each other in Section 3. The section is summed up with an example analysis. In Section 4 we introduce the TinyARM assembly language we design our analysis to run on and discuss how we can represent a TinyARM program as a Control Flow Graph (CFG). The design and implementation of our analyses is covered in Section 5, first by a description of an interval analysis, then how we build the bitflip analysis around it. In Section 6 we present the results of running our analyses on a few interesting code snippets, including something similar to the PAM code above. Additionally, we show a few other features of the tool we have implemented. Finally, we conclude the report in Section 7.

2 Motivation

In our previous work, we looked at existing research in the field of protecting against SEUs or bitflips, as we refer to them as [1]. We found that a multitude of schemes exist that aim to lessen the effect of bitflips on a program simply through the addition of extra instructions. Additionally, specialised or redundant pieces of hardware have been used as protection in both aviation and spaceflight for decades [3]. We found that much of the research in the field follows a pragmatic approach, often built around the idea that bitflips are so exceedingly rare that lowering their damaging effects by one or more orders of magnitude, offers sufficient protection against them in most real world situations [4, 5]. While not an incorrect notion, we took a special interest in the techniques that use formal methods to provide more thorough guarantees such as the type system by Perry et al. [6] or the blue/green encoding by Hansen et al. [7]. In this report we explore how static program analysis can be used to reason about the behaviour of programs in the presence of bitflips. Specifically, we are interested in creating a novel data flow analysis that can show what happens to the possible values in a program under the assumption that a bitflip can occur just before any instruction is executed, but only once during a program's execution. The hope is that the analysis can help identify critical code sections that are vulnerable under this fault model.

In order to better capture when and where an SEU actually occurs in a program, we believe it to be beneficial to analyse programs when they are on a form that most closely resembles how a processor runs them, an assembly language. In our previous work we have presented a definition of the syntax and structural operational semantics of the language TinyARM [1]. We use this same language as the basis for our analyses. We choose to model the possible values stored in registers as intervals both because it gives us better control of how precise the analysis becomes and because that form of analysis is relatively common, which gives us a good starting point.

For our bitflip analysis to be considered useful, it must show that at least some of the protection schemes we review in our previous work are helpful in mitigating the effects of bitflips. A good example is that the robust assert gadget (shown in Listing 2) presented by Hansen et. al [7] has been proven to protect against a broader fault model than the one we use, and as such we expect our analysis to corroborate their findings.

```
    subs r1, r2, r3
    b_ne fail
    cmp r1, #0
    b_ne fail
    ...
```

Listing 2: Robust assert gadget, branches to `fail` if values in r_2 and r_3 are not equal, even if a bitflip occurs

These points are further elaborated in later sections. First we present some background theory about data flow analyses.

3 Data Flow Analysis

In this section we give a cursory overview of what data flow analyses are, discuss some possible applications of them and how a monotone framework can be used to ensure that a data flow analysis always terminates. Finally, we summarise this through a simply example analysis.

3.1 Program Analysis

Static program analyses can be used to prove certain properties about programs without actually running them, which can be beneficial when trying to optimise the code that a compiler generates or used to help a developer pinpoint where they might have introduced errors. For example, recognising that an expression inside a loop never changes means that the evaluation of it can be moved outside the loop without running the risk of changing the semantics of the program. Knowledge about the maximum value a variable can be assigned, help decide how to represent the data at runtime, lowering the overall memory footprint of a program. Knowing that some function can never be reached allows it to be excluded entirely from the compiled code. Additionally, being able to check that all variables are initialised before use or that pointers to freed memory are never used again, makes static program analysis a powerful tool for asserting some important facts about programs. Coupled with integrated development environments some analyses can be run continuously in the background and provide programmers with feedback on their code.

A common way to statically analyse programs, is to operate on a graph representation of the program, often in the form of a CFG. An analysis traverses the CFG and updates the analysis result for each node encountered, based on the results in the surrounding nodes. This means that whenever the results on a node changes from an update, it is necessary to update all its neighbours again, as their result could potentially change given the new input. Intuitively then, the analysis is completed whenever no changes occur when updating nodes anywhere in the graph. It is obvious how relying on such a condition might lead to a non-terminating analysis, but as it turns out, careful definitions of the update functions and the form of the analysis results can guarantee termination, as discussed in later subsections.

3.2 Lattices

In his seminal work, Gary Kildall first explored how transfer functions can be used in conjunction with lattices and CFGs for static program analysis in what has been further generalised and become known as data flow analysis [8, 9].

A lattice is based on a partially ordered set, which is a set, S, and an binary relation, $\sqsubseteq$, for which the following holds:

$$\begin{array}{ll}
\text{Reflexivity:} & \forall x \in S : x \sqsubseteq x \\
\text{Transitivity:} & \forall x, y, z \in S : x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z \\
\text{Anti-symmetry:} & \forall x, y \in S : x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y
\end{array}$$

Similarly, a lattice is a pair consisting of a non-empty set (known as a carrier set), S, and a partial ordering, $\sqsubseteq$, but with the added requirement that all pairs of elements in S have a least upper bound (also known as a join or a supremum) and greatest lower bound (also known as a meet or an infimum) [10, page 34]. To define these, we first need to introduce the notion of upper and lower bounds: We say that an element of some set, is an upper bound for a subset of that same set, if it is greater than or equal to all elements in the subset. Conversely, the element is a lower bound if it is less than or equal to all elements in the subset. Formally:

$$\begin{array}{ll}
\text{Upper bound:} & X \sqsubseteq y \iff \forall x \in X : x \sqsubseteq y \\
\text{Lower bound:} & y \sqsubseteq X \iff \forall x \in X : y \sqsubseteq x
\end{array}$$

where S is a set, $X \subseteq S$ and $y \in S$.

Furthermore, an element is the least upper bound of some subset if it is the smallest upper bound of all the upper bounds of that subset. Dually, an element is the greatest lower bound if it is the largest of all lower bounds of that subset.

$$\begin{array}{ll}
\text{Least upper bound:} & y = \bigsqcup X \iff X \sqsubseteq y \land \forall z \in \{b \mid b \in S, X \sqsubseteq b\} : y \sqsubseteq z \\
\text{Greatest lower bound:} & y = \bigsqcap X \iff y \sqsubseteq X \land \forall z \in \{b \mid b \in S, b \sqsubseteq X\} : z \sqsubseteq y
\end{array}$$

where S is a set, $X \subseteq S$ and $y \in S$.

Occasionally, we will make use of an infix version of the two operators, when the subset is simply two elements:

$$\begin{array}{ll}
\text{Least upper bound:} & x \sqcup y := \bigsqcup \{x, y\} \\
\text{Greatest lower bound:} & x \sqcap y := \bigsqcap \{x, y\}
\end{array}$$

A more restrictive definition of a lattice is that a least upper bound and greatest lower bound must exist for all subsets of the carrier set. Such a lattice is called *complete*, and in fact all finite lattices are complete lattices. To see why this is true, observe that the least upper bound of a subset can be decomposed into a repeated application of least upper bound as the following:

$$\bigsqcup \{x, y, z\} = \bigsqcup \{x, y\} \sqcup z = x \sqcup y \sqcup z$$

Combine this with the fact that the largest subset of a lattice is the carrier set itself, it is clear that for finite lattices this decomposition can be used to obtain a finite number of binary least upper bounds, which eventually evaluates to the greatest element in the carrier set. The same reasoning applies to the greatest lower bound. This does not mean that only finite lattices are complete, however. Consider the infinite lattice $(\mathbb{N}, \leq)$ (all natural numbers ordered under less than or equal), clearly $\bigsqcup \mathbb{N}$ is not defined, as there is no greatest element in $\mathbb{N}$. In this case we can obtain a complete infinite lattice by including ∞ in the carrier set and the ordering, such that it is greater than all members of $\mathbb{N}$. The same reasoning applies to greatest lower bounds and adding a least element to a lattice. The greatest and least elements of a lattice are referred to as *top* ($\top$) and *bottom* ($\bot$) respectively. Note that a bounded infinite lattice is not necessarily a complete lattice.

Adding a bottom element to a lattice and including it in the ordering is an operation that we take advantage of multiple times in our analyses. We therefore define the function *lift* such that it performs this operation:

$$\begin{aligned}&\mathit{lift} : \mathcal{L} \rightarrow \mathcal{L}\\ &\mathit{lift}(S, \sqsubseteq) = (S \cup \{\bot\}, \sqsubseteq \cup \{(\bot, s) \mid s \in (S \cup \{\bot\})\})\end{aligned}$$

where $\mathcal{L}$ is the set of all lattices.

3.3 Lattices in Data Flow Analyses

A lattice is a useful concept for program analyses, as modelling the analysis result of each node in the CFG as an element in a lattice gives us a systematic way of combining results from different paths through a program using either the join or meet functions depending on the type of analysis. Joining two elements means moving towards the top of the lattice, which represents a less precise analysis result, while two elements meeting means moving towards the bottom and more precise results. Perhaps counter intuitively, it is not the goal of all analyses to move toward more precise results, in fact we can distinguish between two types of analyses, those that over-approximate answers and those that under-approximate. We refer to over-approximating analyses as *may* analyses, as the result of them is information that may be true at any given program point. This is achieved by combining information from neighbouring program points using a join. Under-approximating analyses are referred to as *must* analyses, as they result in information that must necessarily be true. For these, the meet operation is used to combine information from neighbouring program points. We can further distinguish between *forward* and *backward* data flow analyses. In the former, the information obtained at each program point (node n in the CFG) is based on information from past program points (predecessors of n), while the latter relies on information from future program points (descendants of n).

For some analyses an identical lattice is used for all programs, but for others the lattice is based on the specific program being analysed. In the latter case, it is common to use a so-called powerset lattice, which is based on a set of properties from the program. The powerset lattice for a finite set, S, ordered under subset can simply be defined as $(\mathcal{P}(S), \subseteq)$. Figure 1 shows the powerset lattice for the set of four integers, $(\mathcal{P}(\{0, 1, 2, 3\}), \subseteq)$. For this lattice, top and bottom are simply $\top = S$ and $\bot = \emptyset$, while the join and meet functions are set-union ($\cup$) and set-intersection ($\cap$) respectively.

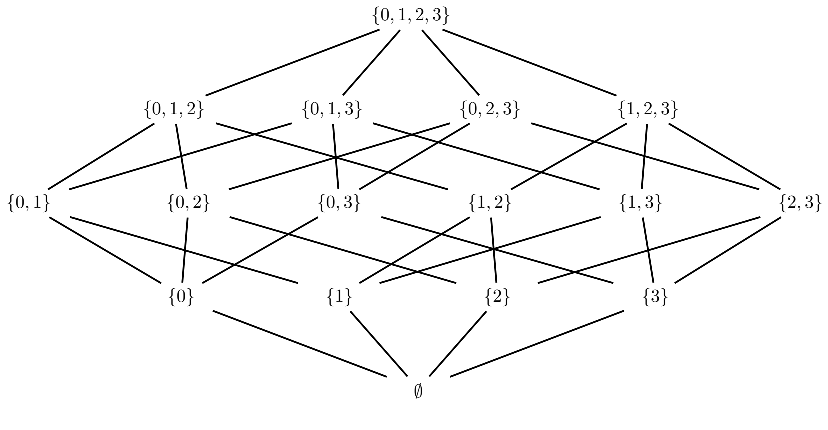

Figure 1: The powerset lattice of $\{0, 1, 2, 3\}$ ordered under subset

Regardless of which lattice we use for a given analysis, we must also define how the analysis result is updated at each node, n, in the CFG. Conceptually, we need a function that outputs a lattice element, by first combining the results from the nodes that lead to n and then performing some operation on that combined information. There are multiple ways of defining functions for performing these operations for a given analysis, the following is simply the way we use when defining analyses in this report. For clarity, we split these two responsibilities into two distinct functions of a CFG-node, *in* and *out*. This allows the definition of *in* to rather elegantly capture the notion of the four types of analyses previously discussed:

$$\begin{array}{ll}
\text{Forward may:} & \mathit{in}(n) = \bigsqcup\{\mathit{out}(p) \mid p \in \mathit{pred}(n)\} \\
\text{Forward must:} & \mathit{in}(n) = \bigsqcap\{\mathit{out}(p) \mid p \in \mathit{pred}(n)\} \\
\text{Backward may:} & \mathit{in}(n) = \bigsqcup\{\mathit{out}(s) \mid s \in \mathit{succ}(n)\} \\
\text{Backward must:} & \mathit{in}(n) = \bigsqcap\{\mathit{out}(s) \mid s \in \mathit{succ}(n)\}
\end{array}$$

where *pred* and *succ* are functions that return the predecessors and successors of a node, respectively.

This leaves *out* with the responsibility of updating the information based on the input from the neighbouring nodes. However, as it is often necessary to update the results differently depending on the type of the node we simply define *out* such that it redirects to the correct transfer function for its input node:

$$\mathit{out}(n) = \begin{cases} \mathit{transfer}_{t_1}(\mathit{in}(n)) & \text{if } \mathit{typeOf}(n) = t_1 \\ \mathit{transfer}_{t_2}(\mathit{in}(n)) & \text{if } \mathit{typeOf}(n) = t_2 \\ \vdots & \\ \mathit{transfer}_{t_m}(\mathit{in}(n)) & \text{if } \mathit{typeOf}(n) = t_m \end{cases}$$

where *typeOf* is an imaginary function that is able to extract information about which type of node it is given.

A transfer function is then defined for each type of node such that it models the update to the analysis result that occurs when the node is executed. In order to guarantee that a data flow analysis terminates, it is crucial that the transfer functions are well behaved, as discussed next.

3.4 Monotone Frameworks

So far we have discussed details about how to structure a data flow analysis, without covering *why* we require this structure. It all boils down to the fact that we want to be able to prove that a given analysis actually terminates, which is only possible if functions applied to the nodes in the CFG eventually reaches a fixed-point, where the output of the function is the same as the input, i.e. $f(x) = x$. As it turns out, Tarski's fixed-point theorem states that if $L = (S, \sqsubseteq)$ is a complete lattice and $f : S \to S$ is an increasing function then P, the set of all fixed-points of f, is also a complete lattice [11]. An increasing function, $f : A \to A$ or more generally $f : B \to C$, where $B \subseteq A$ and $C \subseteq A$, for which $x, y \in B, x \leq y \implies f(x) \leq f(y)$ holds true. This property is also known as order preserving or monotonic. The fact that the set of fixed-points of f is a complete lattice is the crucial detail here, as a lattice by definition cannot be empty. Therefore the theorem actually guarantees that as long as our transfer functions are operating on a complete lattice, repeatedly applying them will always lead to a fixed-point.

This guarantee can be leveraged to create a worklist algorithm, which in broad terms operates by recursively calling itself with a list of nodes that need to be updated: a *worklist*, and the current result of the analysis on all nodes in the CFG: what we call an *out-map*. Each recursive call takes a node off the list, calls the *out* function on it and checks whether the result is different than the previous result stored in the out-map. If it is, the out-map is updated with the new result and all nodes that depend on the current node are added to the worklist. Alternatively, the update made no change to the result, and we simply proceed to update the next node in the worklist. Given what we know about fixed-points, it is clear that the worklist will eventually be empty and we are left with an out-map containing the result of the analysis.

The pseudo code based on Haskell conventions shown in Listing 3 attempts to capture this notion:

```
worklist :: [Node] -> Map Node LatticeElements
worklist [] outMap = outMap
worklist (node:rest) outMap =
    if out node == (outMap ! node)
        then worklist rest outMap
        else worklist (rest ++ (dependants node)) (insert node (out node) outMap)
```

Listing 3: Pseudo code for a worklist algoritm

3.5 Live Variables Analysis: An Example

In this subsection we summarise what we have covered about data flow analyses by defining a rather simple analysis and running it on a small program. The analysis we construct is a so-called live variables analysis, the result of which for any given program point is the set of live variables. If a variable is live, it means that it is possible for it to be read somewhere later in the program without it being written to in between. Knowing this property is useful for a compiler, as it can lead to a program which makes better use of the limited number of registers available, as storing non-live variables can be avoided.

Figure 2 shows the small C-like program we run the example analysis on and a possible CFG representation of it, where each node represents a single expression in the program.

```
int z = 20
int x = 10
int y = x + z
if (y == 42)
  print(x)
else
  print(y)
```

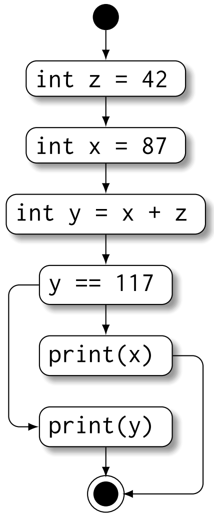

Figure 2: Example program as code and CFG

How a CFG is defined also has an impact on how an analysis functions. For example, the way we have created one here, means the result of a node is the same on all its outgoing edges. For this type of analysis this does not matter, but as discussed in Subsection 4.3 this is not always the case.

The result at a given program point is going to be the set of variables that *may* be live at that point. We therefore use a powerset lattice to represent the analysis results, specifically the powerset lattice of the set of all variables in the program: $(\mathcal{P}(\{x, y, z\}), \subseteq)$. Furthermore, whether a variable is live or not at a given program point depends on whether or not it is used at a future program point, which means we are dealing with a backward analysis. Therefore we can define our *in* function as $in(n) = \bigsqcup\{out(s) \mid s \in succ(n)\}$. In order to define the *out* function we need to reason about which transfer functions we need. Starting from the end of the program, no variables are live. Whenever we encounter a variable being used we add it to the analysis result, and only remove it again once we encounter a node which assigns to it. The *out* function simply needs to distinguish between those two cases and call separate transfer functions for them, but in contrast to our previous definition of the *out* function, the transfer functions require additional information. This is a common occurrence, and moving forward our transfer functions require quite a bit of extra information. Additionally, seeing as a single node can contain both an assignment and a use, as seen in the third line of the example program, the transfer function for assignments also needs to distinguish between the left and right side of the node:

$$out(n) = \begin{cases} transfer_{assign}(in(n), left(n), vars(n)) & \text{if } typeOf(n) = assignment \\ transfer_{expression}(in(n), vars(n)) & \text{if } typeOf(n) = expression \end{cases}$$

$$transfer_{assign}(liveVars, leftVar, rightVars) = liveVars \setminus \{leftVar\} \cup rigthtVars$$

$$transfer_{expression}(liveVars, vars) = liveVars \cup vars$$

where *left* is function that extracts the name of the variable being assigned to and *vars* is a function that extracts the names of the variables being used in a node.

Clearly $tranfer_{expression}$ is monotonic with respect to the lattice ordering, as adding the same thing to two sets cannot possibly change which one of them is a subset of the other - at the most, they become equal. The same goes for removing the same thing from two sets, which also makes $transfer_{assign}$ monotonic. Seeing as both transfer functions are monotonic, we can run the analysis using a worklist algorithm and be guaranteed that it terminates. When it does we are left with the results shown on the CFG in Figure 3.

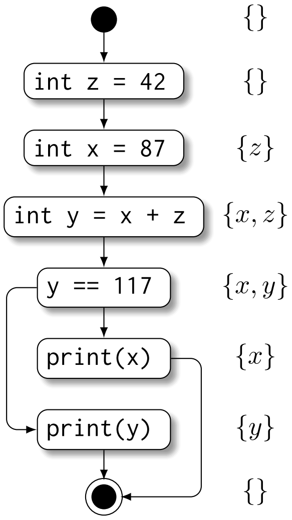

Figure 3: Result of live variable analysis on the example program

4 TinyARM

As mentioned in Section 2, it really only makes sense to do an analysis relating to bitflips on a level as close to the hardware where they occur as possible. The instruction set of ARM is simpler than that of the x86 architecture and is therefore a bit easier to model. However, it still contains a significant number of instructions, many of which are of limited use in our situation. To keep these extraneous instructions from muddling the picture, we choose to only use a subset of ARM. We have already presented a formal definition of the syntax and structural operational semantics of the language TinyARM in our previous work [1]. TinyARM is a well defined simplification of ARM and very closely related to the language of the same name introduced by Hansen et al. [7]. In the following subsections we reintroduce a few key concepts of TinyARM that we utilise for our analyses and discuss how we turn a TinyARM program into a CFG. The definition of the language in its entirety can be found in Appendix A.

4.1 Semantics and Fault Model

Our previous definition of TinyARM did not allow for constant values as the last parameter to addition, subtraction and comparison instructions. This was a deliberate choice, as we avoided superfluous instructions in an attempt to simplify the language. Given that our analyses force us to actually use the language to write programs, we choose to introduce additional instructions for these operations. Additionally, we adopt the convention of prefixing all constants with a # symbol. Formally, we are adding the following instructions to the language:

$$\begin{array}{rlll} \mathit{instr}_{\mathit{cons}} & ::= & \texttt{ADD}_{\chi}\ x, y, v & \text{add value in } y \text{ to value } v \text{ and store result in } x \\ & \mid & \texttt{ADDS}_{\chi}\ x, y, v & \text{same as } \texttt{ADD}\text{, but also set flags} \\ & \mid & \texttt{SUB}_{\chi}\ x, y, v & \text{subtract value } v \text{ from value in } y \text{ and store result in } x \\ & \mid & \texttt{SUBS}_{\chi}\ x, y, v & \text{same as } \texttt{SUB}\text{, but also set flags} \\ & \mid & \texttt{CMP}_{\chi}\ x, v & \text{compare value in } x \text{ with value } v \text{ and set flags} \end{array}$$

The new semantic rules are very similar to the existing ones so we only include one here, while the remaining can be found in Appendix A.

$$[\mathrm{cmp}_{\mathrm{val}}] \quad \frac{P(R(r_{pc})) = \texttt{CMP}_{\chi}\ x, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F \rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, \mathit{flags}_{\texttt{SUB}}(R(x), v) \rangle}$$

Our previous formal description of TinyARM is used to give precise definitions of multiple different fault models, such as SEUs being able to occur in heap memory, program code and flags. However, in the analysis we are designing here, we are only operating under one of those fault models, which simply allows one SEU to occur in a general purpose register. The SEU is able to flip a single bit in the register, i.e. change a 0 to a 1 or a 1 to a 0. In our previous work this fault models was known as $[f\text{-reg}_{\mathrm{gen}}]$ and defined as:

$$\frac{x \in \mathrm{GeneralRegister} \qquad v = R(x) \qquad v' \equiv_1 v}{\langle P, H, R, F \rangle \Longrightarrow_{f\text{-reg}_{\mathrm{gen}}} \langle P, H, R[x \mapsto v'], F \rangle}$$

See Appendix A for the supporting definitions.

4.2 Conditional Instructions

Under the assumption that a maximum of one SEU occurs during a program's execution, repeated comparisons is a possible defence against changes to a register's value going unnoticed, as shown in Section 2. Given this fact, it is important for our analysis to extract as much information from comparisons as possible. A way to accomplish this, is by tracking the changes to the four control flags of TinyARM. TinyARM, like the original ARM architecture makes use of condition codes to achieve conditional execution. The following definitions are mostly copied directly from our previous work, except for the exclusion of the condition code `NV`, as it was strictly included to solve a problem that is not relevant here. Each instruction can be annotated with a condition code, and the instruction is only executed if the condition holds. Whether a condition holds or not, depends on the current state of the four control flags: *Negative*, *Zero*, *Carry* and *Overflow*:

$$\begin{aligned}&\text{Flag} = \{f_N, f_Z, f_C, f_V\}\\ &\textbf{Flags} = \text{Flag} \to \mathbb{B}\end{aligned}$$

where $\mathbb{B} = \{0, 1\}$ and the bold marking indicates a set of functions. Furthermore, $\mathbb{B}_n = [0..n-1] \to \mathbb{B}$, with 0 referring to the least significant bit.

Each flag has a binary state that is set by specific instructions depending on their operands. The condition codes we use are as follows:

$$\text{ConditionCode} = \{\texttt{EQ}, \texttt{NE}, \texttt{CS}, \texttt{CC}, \texttt{MI}, \texttt{PL}, \texttt{VS}, \texttt{VC}, \texttt{HI}, \texttt{LS}, \texttt{GE}, \texttt{LT}, \texttt{GT}, \texttt{LE}, \texttt{AL}\}$$

The function *cond* allows us to determine whether or not an instruction with a given condition code, χ, should be executed given some state of the control flags.

$$\mathit{cond}\colon \text{ConditionCode} \times \textbf{Flags} \to \{\text{true}, \text{false}\}$$

$$\mathit{cond}(\chi, F) = \begin{cases} F(f_Z) = 1 & \text{if } \chi = \texttt{EQ}\\ F(f_Z) = 0 & \text{if } \chi = \texttt{NE}\\ F(f_C) = 1 & \text{if } \chi = \texttt{CS}\\ F(f_C) = 0 & \text{if } \chi = \texttt{CC}\\ F(f_N) = 1 & \text{if } \chi = \texttt{MI}\\ F(f_N) = 0 & \text{if } \chi = \texttt{PL}\\ F(f_V) = 1 & \text{if } \chi = \texttt{VS}\\ F(f_V) = 0 & \text{if } \chi = \texttt{VC}\\ F(f_C) = 1 \land F(f_Z) = 0 & \text{if } \chi = \texttt{HI}\\ F(f_C) = 0 \lor F(f_Z) = 1 & \text{if } \chi = \texttt{LS}\\ F(f_N) = F(f_V) & \text{if } \chi = \texttt{GE}\\ F(f_N) \neq F(f_V) & \text{if } \chi = \texttt{LT}\\ F(f_Z) = 0 \land F(f_N) = F(f_V) & \text{if } \chi = \texttt{GT}\\ F(f_Z) = 1 \lor F(f_N) \neq F(f_V) & \text{if } \chi = \texttt{LE}\\ \text{true} & \text{if } \chi = \texttt{AL} \end{cases}$$

In TinyARM there are a total of three instructions which can set the flags: `ADDS`, `SUBS`, and `CMP`, with the two latter instructions being semantically similar, as a comparison is a subtraction where the resulting difference is discarded. The following is a quick overview of the intuitive meaning of each flag: The f_N flag is set whenever the result of an operation has the most significant bit set, as this means the result is negative using two's complement. The f_Z is simply set whenever the result is exactly zero. The f_C is set whenever the result of an addition requires an additional bit to be set. This also works for subtraction, as an addition still takes place under the hood. The intuitive way to think of the carry flag for subtraction, is that it is set when the *real* (treating the operands and result as regular, unbounded integers) result of the subtraction is non-negative. The final flag, f_V is used to indicate that an over- or underflow has occurred. For addition the flag is set when the sum of two positive numbers is negative, or when the sum of two negative numbers is positive.

When subtracting, the flag is set when a negative number subtracted from a positive number yields a negative result or when a positive number subtracted from a negative number yields a positive result. For a more comprehensive formalisation of how each flag is set we again refer to our previous work, or the full definitions found in Appendix A.

4.3 Control Flow Graph Creation

Seeing as we are interested in conditional branching giving our analysis more precision, we require a way for the analysis to distinguish between the different state of the control flags that arise when using conditionals. For example, in the few lines of code seen in Figure 4 we would like for our analysis to recognise that at line 4 the value stored in r_1 must necessarily be 87, while it must be different from 87 if control is transferred to the `failure` label.

```
1       ldr r1, #42
2       cmp r1, #87
3       b_ne failure
4       str r2, #42
```

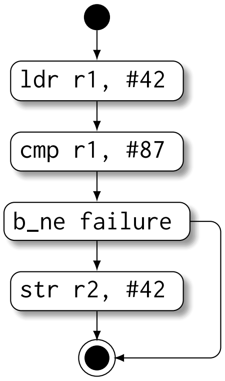

Figure 4: TinyARM example program and a simplistic CFG

The property we want our analysis to have, is known as path-sensitivity. One way of achieving this, is by having the analysis operate on edges in the CFG instead of nodes, as this means the two outgoing edges on the branch node are entirely separate from the analysis' point of view. However, we choose to solve the problem by introducing additional nodes in the CFG, such that every time a conditional instruction is encountered we add two nodes, an *assert* and a *refute* node. Separate transfer functions can then be defined for each node. This leads to the CFG shown in Figure 5.

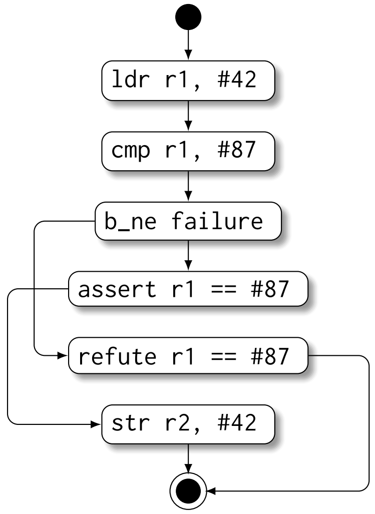

Figure 5: TinyARM CFG with assert and refute nodes

While this solves the problem for conditional branching, other conditional instructions have a problematic representation with this CFG. As can be seen in Figure 6, the actual instruction that is conditionally executed is located earlier in the CFG than the assert node, whose output is the analysis result we want as the input of the conditional instruction.

```
    ldr r1, #42
    cmp r1, #87
    str_eq r2, #42
```

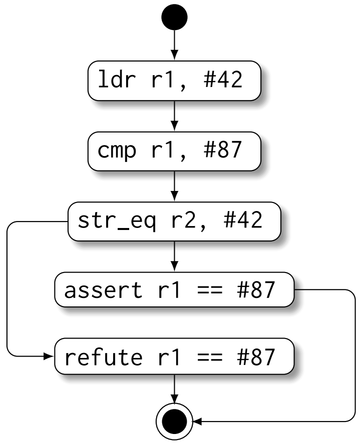

Figure 6: Example program with a conditional instruction and a CFG with assert and refute nodes

While it is certainly possible to deal with this problem in the definitions of the transfer functions, we choose to model the CFG differently to better capture the underlying branching that takes place when a conditional instruction is encountered. We accomplish this by positioning the instruction such that it is a child of the assert node and introduce an *if* node in its place. This leads to the final structure of CFGs as used by our analyses, as shown in Figure 7

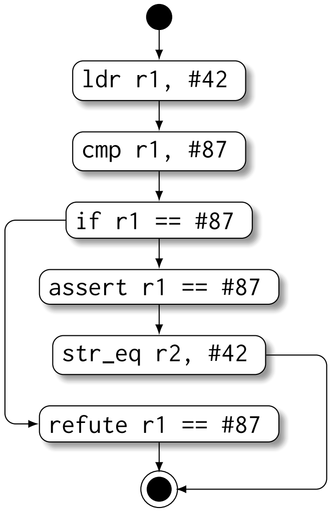

Figure 7: Final CFG structure for TinyARM

Introducing additional nodes in a CFG can negatively impact the runtime of a data flow analysis, but the relatively small programs we expect to run the analysis on and the simpler definition of the analysis is worth it, in our opinion. To reduce the size of CFGs somewhat, consecutive instructions using the same condition code are grouped under one assert node.

5 Design and implementation

In this section we present our two analyses, interval and bitflip, in detail. Our implementation is made in Haskell, but we mainly present the analyses through formal mathematical descriptions. The definitions of both analyses in their entirety can be found in Appendices B through C.

5.1 Interval Analysis

As mentioned in Section 2, we build our bitflip analysis around a more traditional analysis: an interval analysis. While intervals sound less precise than single values, in actuality they can be made just as precise as a single value can simply be represented by a singleton interval. For our analysis we are giving up some precision and using the intervals to over approximate the values stored in registers. A way in which our analysis differs from a traditional interval analysis, is by the fact that we model the control flags, such that any interval is accompanied by a four-tuple which describes the state of the individual flags using an interval lattice. We define an interval lattice L_{I_n} such that it is parameterised in the maximum value it can contain, n:

$$L_{I_n} = \mathit{lift}((\{[a..b] \mid a, b \in \mathbb{N}_n, a \leq b\}, \sqsubseteq_{I_n}))$$
$$\sqsubseteq_{I_n} = \{([l_1..r_1], [l_2..r_2]) \mid [l_1..r_1], [l_2..r_2] \in I_n, l_1 \geq l_2, r_1 \leq r_2\}$$

where $\mathbb{N}_x = \{n \mid n \in \mathbb{N}, n \leq x\}$.

The interval lattice is ordered under inclusion with singleton intervals just above a synthetic $\bot$ element, which conceptually represents an undefined interval. Moving up through the lattices, each level is made up of ever fewer intervals with an increasing number of elements in them until the largest interval is found at the top, $\top = [0..n]$. Figure 8 illustrates this lattice structure for an interval lattice with $n = 2^8 - 1$.

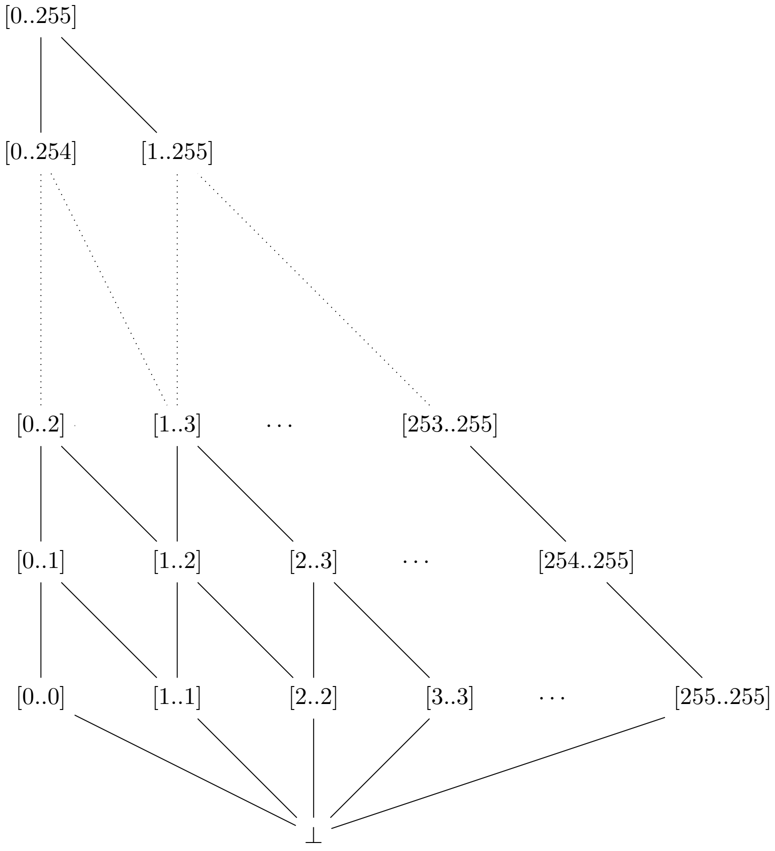

Figure 8: Structure of the interval lattice $L_{I_{255}}$

For the analysis we are interested in capturing the possible values in all general registers, which we model as a set of functions from register names to interval lattices with a max value determined by the bit width of the architecture. For most of our experimentation we have kept to 8 bits as this makes reading and reasoning about the values simpler, but we eventually discuss how our analyses can be run on arbitrary bit widths.

$$Vars = GeneralRegister \rightarrow L_{i_{MAX}}$$

where MAX is a constant that is available globally and is based on the bit width, bw. For the examples in this section $bw = 8$ and therefore $MAX = 2^8 - 1$.

These definitions would be enough to define the lattice for a traditional interval lattice, but as previously mentioned, we are interested in the extra precision we can achieve by modelling the control flow using the control flags. We can simply model each binary control flag as an interval lattice with a max value of 1:

$$L_f = L_{I_1} \times L_{I_1} \times L_{I_1} \times L_{I_1}$$

This leaves us with the final lattice for an interval analysis, L:

$$L = (\mathcal{P}(L_f \times Vars), \subseteq)$$

An element in L is a set of possible configurations of control flags with an associated interval that describes the possible values stored in each general purpose register. The following example shows an excerpt of a single lattice element:

$$\begin{array}{l} \{ \\ \quad (([0..0],[1..1],[1..1],[0..0]),[r_1 \mapsto [42..42], r_2 \mapsto [42..42], \ldots]), \\ \quad (([0..0],[0..0],[0..0],[0..0]),[r_1 \mapsto [0..41], r_2 \mapsto [43..100], \ldots]), \\ \quad \ldots \\ \} \end{array}$$

As an interval analysis is a forward may analysis, our *in* function is defined as:

$$\mathit{in}(n) = \bigsqcup\{\mathit{out}(p) \mid p \in \mathit{pred}(n)\}$$

By abusing notation a slight bit, we make the equality operator on nodes perform a kind of pattern matching. This leaves us with the following definition of the *out* functions:

$$\mathit{out}(n) = \begin{cases} t_{\texttt{MOV}v}(r, v, \mathit{in}(n)) & \text{if } n = \texttt{MOV}_\chi\ r, v \\ t_{\texttt{MOV}r}(r_{dst}, r_{src}, \mathit{in}(n)) & \text{if } n = \texttt{MOV}_\chi\ r_{dst}, r_{src} \\ t_{\texttt{LDR}}(r, \mathit{in}(n)) & \text{if } n = \texttt{LDR}_\chi\ r, \_ \\ t_{\texttt{ADD}v}(r_{dst}, r_{src}, v, \mathit{in}(n)) & \text{if } n = \texttt{ADD}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{ADD}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{in}(n)) & \text{if } n = \texttt{ADD}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{ADDS}v}(r_{dst}, r_{src}, v, \mathit{in}(n)) & \text{if } n = \texttt{ADDS}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{ADDS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{in}(n)) & \text{if } n = \texttt{ADDS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{SUB}v}(r_{dst}, r_{src}, v, \mathit{in}(n)) & \text{if } n = \texttt{SUB}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{SUB}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{in}(n)) & \text{if } n = \texttt{SUB}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{SUBS}v}(r_{dst}, r_{src}, v, \mathit{in}(n)) & \text{if } n = \texttt{SUBS}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{SUBS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{in}(n)) & \text{if } n = \texttt{SUBS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{CMP}v-}(r, v, \mathit{in}(n)) & \text{if } n = \texttt{CMP}_\chi\ r, v \\ t_{\texttt{CMP}r-}(r_x, r_y, \mathit{in}(n)) & \text{if } n = \texttt{CMP}_\chi\ r_x, r_y \\ t_{\texttt{ASSERT}}(\chi, \mathit{in}(n)) & \text{if } n = \texttt{ASSERT}\chi \\ t_{\texttt{REFUTE}}(\chi, \mathit{in}(n)) & \text{if } n = \texttt{REFUTE}\chi \\ \mathit{in}(n) & \text{otherwise} \end{cases}$$

We discuss the definitions of a few of these in depth, while the remaining definitions can be found in Appendix B.

Typically data flow analyses use a starting state that assumes all CFG nodes to be $\bot$, which for an analysis based around a powerset means the empty set, $\emptyset$. However, the way our transfer functions are defined, means they expect a certain structure to be present in the states they operate on. Therefore we initialise all nodes with the following default state, in which all four flags are $\bot$ and all registers map to $\bot$:

$$\{((\bot,\bot,\bot,\bot),(\lambda(r).\bot))\}$$

While the same can possibly be achieved by modifying the lattice or transfer functions, we find that this solution leaves us with the most elegant transfer functions.

The `LDR` instruction loads a value from the heap into a register. Since we have no way of knowing what value is stored at the memory location, the corresponding transfer function, $t_{\texttt{LDR}}$, maps to a new state where the destination register maps to ⊤ while the control flags and remaining registers are left unchanged.

$$\begin{aligned}&t_{\mathsf{LDR}} : \mathit{GeneralRegister} \times L \to L\\&t_{\mathsf{LDR}}(r, l) = \{(f, i[r \mapsto [0..\mathit{MAX}]]) \mid (f, i) \in l\}\end{aligned}$$

Clearly t_{LDR} is monotonic with respect to the subset ordering of the powerset lattice, as any change it makes, is contained in the individual tuples of lattice elements - it neither adds nor removes tuples to the set. In fact this is true for most of our transfer functions. Turning to the transfer functions for the instructions which set the control flags, this is no longer the case, as these functions can produce states with more tuples, as they split intervals depending on the state of their flags.

The following definition of the transfer function for comparisons between two registers $t_{\mathsf{CMP}r-}$ relegates the interesting part, the splitting, to the function *split*.

$$\begin{aligned}&t_{\mathsf{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L\\&t_{\mathsf{CMP}r-}(r_x, r_y, l) = \{(f', i[r_x \mapsto a][, r_y \mapsto b]) \mid (\_, i) \in l, (f', a, b) \in \mathit{split}_{\mathsf{sub}}(i(r_x), i(r_y))\}\end{aligned}$$

Ignoring the call to *combine*, the function $split_{sub}$ simply generates a set of triples of flags and two singleton intervals. This is accomplished by enumerating all the possible combinations of values in its two input intervals and calling $\mathit{flags}_{\mathsf{SUB}}$ on each of them. Clearly, this enumeration of all values in two intervals can become a computationally expensive task, especially using a larger bit width. In Subsection 5.2 we discuss the implications of this more thoroughly and offer a more efficient solution, which produces the same results.

$$\begin{aligned}&\mathit{split}_{\mathsf{sub}} : L_{I_{MAX}} \times L_{I_{MAX}} \to \mathcal{P}(L_f \times L_{I_{MAX}} \times L_{I_{MAX}})\\&\mathit{split}_{\mathsf{sub}}(i_1, i_2) = \mathit{combine}(\{((f(f_N), f(f_Z), f(f_C), f(f_v)), [a..a], [b..b]) \mid a \in i_1, b \in i_2, f = \mathit{flags}_{\mathsf{SUB}}(a, b)\})\end{aligned}$$

The following is an example of the output *without* the *combine* function:

$$\begin{aligned}&\mathit{split}_{\mathsf{sub}}([41..43], [41..42]) = \{\\&\quad(([0..0], [1..1], [1..1], [0..0]), [41..41], [41..41]]),\\&\quad(([1..1], [0..0], [0..0], [0..0]), [41..41], [42..42]]),\\&\quad(([0..0], [0..0], [1..1], [0..0]), [42..42], [41..41]]),\\&\quad(([0..0], [1..1], [1..1], [0..0]), [42..42], [42..42]]),\\&\quad(([0..0], [0..0], [1..1], [0..0]), [43..43], [41..41]]),\\&\quad(([0..0], [0..0], [1..1], [0..0]), [43..43], [42..42]])\\&\}\end{aligned}$$

The *combine* function combines the intervals for which the flags are the same, which leads to a loss of precision:

$$\begin{aligned}&\mathit{combine} : P(L_f \times L_{I_{MAX}} \times L_{I_{MAX}}) \to P(L_f \times L_{I_{MAX}} \times L_{I_{bw}})\\&\mathit{combine}(X) = \{(f, [\mathit{min}(L_1)..\mathit{max}(R_1)], [\mathit{min}(L_2)..\mathit{max}(R_2)]) \mid\\&\qquad L_1 = \{\, l \mid (f, [l..\_], \_) \in X \,\},\\&\qquad R_1 = \{\, r \mid (f, [\_..r], \_) \in X \,\},\\&\qquad L_2 = \{\, l \mid (f, \_, [l..\_]) \in X \,\},\\&\qquad R_2 = \{\, r \mid (f, \_, [\_..r]) \in X \,\}\\&\}\end{aligned}$$

The following is an example of the actual output of the $\mathit{split}_{\mathsf{sub}}$ function:

$$\begin{aligned}&\mathit{split}_{\mathsf{sub}}([41..43], [41..42]) = \{\\&\quad(([0..0], [1..1], [1..1], [0..0]), [41..42], [41..42]]),\\&\quad(([1..1], [0..0], [0..0], [0..0]), [41..41], [42..42]]),\\&\quad(([0..0], [0..0], [1..1], [0..0]), [42..43], [41..42]]),\\&\}\end{aligned}$$

In the first example, it was clear that the singleton intervals [41..41] and [42..42] give the same flag state when compared to themselves, but when combined, we have that the two intervals [41..42] and [41..42] give these same flags, which is not wrong, but it is not clear that comparing 41 and 42 would *not* lead to those flags.

The monotonicity of the $t_{\mathsf{CMP}r-}$ function and other transfer functions that add additional tuples to the analysis result is not immediately obvious. In fact we simply conjecture that it is and present no proof.

As discussed in Subsection 4.3 we are interested in conditionals reducing the number of possible values, such that comparisons can be used to remove states from the analysis result. This is accomplished by the transfer functions t_{ASSERT} and t_{REFUTE}, which simply work by filtering out irrelevant tuples:

$$\begin{aligned}
&t_{\mathsf{ASSERT}} : \mathit{ConditionCode} \times L \to L\\
&t_{\mathsf{ASSERT}}(\chi, l) = \{(f, i) \mid (f, i) \in l, \mathit{matchesCondition}(f, \chi)\}\\
&t_{\mathsf{REFUTE}} : \mathit{ConditionCode} \times L \to L\\
&t_{\mathsf{REFUTE}}(\chi, l) = \{(f, i) \mid (f, i) \in l, \mathit{matchesCondition}(f, \neg\chi)\}
\end{aligned}$$

Both functions are monotone, as they will always remove the same tuples from two results if those two results are comparable.

The *matchesCondition* function is very similar to the function *cond* used in the semantics of TinyARM, except that it operates on interval lattices.

$$\begin{aligned}
&\mathit{matchesCondition} : L_f \times \mathit{ConditionCode} \to \{\mathit{true}, \mathit{false}\}\\
&\mathit{matchesCondition}((n, z, c, v), \chi) =\\
&\quad\begin{cases}
[1..1] \sqsubseteq z & \text{if } \chi = \texttt{EQ}\\
[0..0] \sqsubseteq z & \text{if } \chi = \texttt{NE}\\
[1..1] \sqsubseteq c & \text{if } \chi = \texttt{CS}\\
[0..0] \sqsubseteq c & \text{if } \chi = \texttt{CC}\\
[1..1] \sqsubseteq n & \text{if } \chi = \texttt{MI}\\
[0..0] \sqsubseteq n & \text{if } \chi = \texttt{PL}\\
[1..1] \sqsubseteq v & \text{if } \chi = \texttt{VS}\\
[0..0] \sqsubseteq v & \text{if } \chi = \texttt{VC}\\
[0..0] \sqsubseteq z \land [1..1] \sqsubseteq c & \text{if } \chi = \texttt{HI}\\
[0..0] \sqsubseteq c \lor [1..1] \sqsubseteq z & \text{if } \chi = \texttt{LS}\\
([1..1] \sqsubseteq n \land [1..1] \sqsubseteq v) \lor ([0..0] \sqsubseteq n \land [0..0] \sqsubseteq v) & \text{if } \chi = \texttt{GE}\\
([1..1] \sqsubseteq n \land [0..0] \sqsubseteq v) \lor ([0..0] \sqsubseteq n \land [1..1] \sqsubseteq v) & \text{if } \chi = \texttt{LT}\\
([1..1] \sqsubseteq n \land [0..0] \sqsubseteq z \land [1..1] \sqsubseteq v) \lor ([0..0] \sqsubseteq n \land [0..0] \sqsubseteq z \land [0..0] \sqsubseteq v) & \text{if } \chi = \texttt{GT}\\
([1..1] \sqsubseteq z) \lor ([1..1] \sqsubseteq n \land [0..0] \sqsubseteq v) \lor ([0..0] \sqsubseteq n \land [1..1] \sqsubseteq v) & \text{if } \chi = \texttt{LE}\\
\mathit{true} & \text{if } \chi = \texttt{AL}\\
\mathit{false} & \text{otherwise}
\end{cases}
\end{aligned}$$

The function actually over approximates the conditions that match, given that flags with the interval [0..1] ($\top$) match both as 1 and 0. In actuality this never becomes relevant, as the analysis does not include any way for flags to be joined such that they become $\top$. We experimented with a more aggressive merging of tuples in a way that produces $\top$ flags. This was done by first using a liveness analysis similar to the one described in Subsection 3.5 to obtain knowledge about which specific condition codes are going to be used from an instruction sets the flags until the next instruction that sets the flags. Unfortunately, the merging of the flags, means that the over approximation in *matchesCondition* results in too great a loss of precision.

The transfer function $t_{\mathsf{ADD}v}$ is used for the addition instruction where the third parameter is a constant and is interesting because of how we define addition. Seeing as we are operating on integer values with an upper and lower bound defined by the bit width we have to consider what happens when an addition results in an interval which wraps through zero to the maximum value due to under- or overflow.

$$\begin{aligned}&t_{\mathsf{ADD}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \to \mathbb{N}_{MAX} \times L \to L\\&t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, l) = \{(f, i[r_{dst} \mapsto add_I(i(r_{src}), [v..v])]) \mid (f, i) \in l\}\end{aligned}$$

For example consider addition between the two intervals [1..10] and [250..250] with a bit width of 8. Clearly some of the values are going to produce an overflow, so we would be hard pressed to accurately represent the result as an interval. The most precise way is to split the result into the two intervals [0..4] and [251..255]. Another choice is to simply over approximate and say the result is somewhere between 0 and 255, i.e. ⊤. As can be seen below we choose to implement the latter version due to its simplicity.

$$\begin{aligned}&add_I : L_{I_{MAX}} \times L_{I_{MAX}} \to L_{I_{MAX}}\\&add_I(\bot, \_) = \bot\\&add_I(\_, \bot) = \bot\\&add_I([l_1..r_1], [l_2..r_2]) = \begin{cases}[0..MAX] & \text{if } (l_1 + l_2) \leq MAX \wedge (r_1 + r_2) > MAX\\ [(add_{bin}(l_1, l_2, 0)..add_{bin}(r_1, r_2, 0)] & \text{otherwise}\end{cases}\end{aligned}$$

In cases where the addition always over- or underflows it is possible to accurately represent the result as a single interval. Our implementation catches these cases and the use of the add_{bin} function makes sure that the addition is performed such that the result correctly wraps around.

Finally, each transfer function for addition and subtraction that also sets flags are simply compositions of the function that performs the addition or subtraction and a comparison function. For example, $t_{\mathsf{ADDS}v}$ is composed of $t_{\mathsf{ADD}v}$ and $t_{\mathsf{CMP}v+}$:

$$\begin{aligned}&t_{\mathsf{ADDS}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \to \mathbb{N}_{MAX} \to L \to L\\&t_{\mathsf{ADDS}v}(r_{dst}, r_{src}, v, l) = t_{\mathsf{CMP}v+}(r_{src}, v, t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, l))\end{aligned}$$

This constitutes our interval analysis. Again, the full list of definitions can be found in Appendix B.

5.2 Efficiency of *split*

As previously mentioned, the simple way of calculating the state of the flags, by enumerating all the combinations of values in two intervals, is by no means efficient, especially not on the more realistic bit width of 32, where each interval risks containing billions of values. The implementation of $split_{\mathsf{sub}}$ as the Haskell function `splitSubsBruteForce` can be seen in Listing 4. The time complexity of the function is linear in the product of the lengths of each interval, and as such has a worst case complexity of $O(n^2)$, where n is the largest value of both intervals. Seeing as transfer functions such as t_{LDR} always transfers to states where intervals are as wide as possible, the worst case is actually quite likely to appear in many programs, which makes it practically infeasible to run the analysis with a bit width higher than 16.

```haskell
splitSubsBruteForce :: Interval -> Interval -> [(FlagsLattice, (Interval, Interval))]
splitSubsBruteForce i1@(Interval l1 r1) i2@(Interval l2 r2) =
  combine $ map
    (\(lVal, rVal) -> (evalFlagSubs lVal rVal, (lVal, rVal)))
    [ (lVal, rVal) | lVal <- [l1..r1], rVal <- [l2..r2] ]

evalFlagSubs :: Value -> Value -> FlagsLattice
evalFlagSubs val1 val2 = FlagsLattice flagN flagZ flagC flagV
  where
    wordSize = finiteBitSize val1
    resultWord = val1 + ((complement val2) + 1)
    resultInt = (toInteger val1) - (toInteger val2)
    msb1 = testBit val1 (wordSize - 1)
    msb2 = testBit val2 (wordSize - 1)
    msbR = testBit resultWord (wordSize - 1)
    flagN = if msbR                         then BitTrue else BitFalse
    flagZ = if resultWord == 0              then BitTrue else BitFalse
    flagC = if resultInt < 0                then BitFalse else BitTrue
    flagV = if msb1 /= msb2 && msb1 /= msbR then BitTrue else BitFalse

combine :: [(FlagsLattice, (Value, Value))] -> [(FlagsLattice,(Interval,Interval))]
combine lst = combine’ (L.sort lst) []
  where
    combine’ :: [(FlagsLattice, (Value, Value))] ->
      [(FlagsLattice,(Interval,Interval))] ->
      [(FlagsLattice,(Interval,Interval))]
    combine’ [] lst = lst
    combine’ ((flags, (lval, rval)):rest) [] =
      combine’ rest [(flags, (Interval lval lval, Interval rval rval))]
    combine’ ((flags, (lval, rval)):rest) (prev@(pFlags,(pLInterval, pRInterval)):restResult) =
      combine’ rest $
        if flags == pFlags
          then (flags, (fit lval pLInterval, fit rval pRInterval)) : restResult
          else (flags, (Interval lval lval, Interval rval rval)) : (prev : restResult)

    fit :: Value -> Interval -> Interval
    fit newVal (Interval oldMin oldMax) =
      if newVal <= oldMin
        then Interval newVal oldMax
        else if newVal >= oldMax
          then Interval oldMin newVal
          else Interval oldMin oldMax
```

Listing 4: Haskell implementation of $split_{\text{sub}}$

Through experimentation, we find that `evalFlagSubs` is only capable of producing seven different combinations of control flags. Furthermore, by observing how the resulting intervals output by `splitSubsBruteForce` differ when making slight changes to the input, we find that the results follow a pattern which we can decode. Mostly through trial and error, this leads to the implemented solution of which an excerpt is shown in Listing 5. The splitting of the intervals always occur around a few special values, zero, max signed and max unsigned. The time complexity of the new proposed solution is independent of the lengths of the intervals. How to prove the equivalence of the two functions in a general way is not immediately obvious to us, so we instead prove it by example, by comparing the output of the two functions for all combinations of two intervals. For all bit widths up to and including 8 bits, there are no differences between the outputs of the two functions. For an arbitrary bit width, bw, there are a total of $(\frac{2^{bw}(2^{bw}+1)}{2})^2$ unique intervals to compare. For 8 bits that equates to approximately one billion. As each of those intervals requires calling the `evalFlagSubs` function once per unique combination of values in the two intervals, the number of calls quickly explodes, in fact for 8 bits this number is exactly 8.003.557.851.136. Therefore, we are only able to prove that our efficient function works up to a bit width of 8 and only after several days of computing using the largest virtual CPU available to us at Aalborg University's CLAAUDIA service (32 threads). Our solution does break down with a bit width of 1, but should anyone wish to analyse TinyARM programs on a 1 bit architecture, the brute force method should be sufficient for their needs.

```haskell
splitSubs :: Interval -> Interval -> [(FlagsLattice, (Interval, Interval))]
splitSubs i1@(Interval l1 r1) i2@(Interval l2 r2) =
  let
    al1 = if l2 >= maxS  then maxU else l2+maxS+1
    ar1 = r1
    al2 = l2
    ar2 = if r1 <= maxS  then 0    else r1-maxS-1

    bl1 = if l2 <= maxS  then 0    else l2-maxS-1
    br1 = maxS
    bl2 = maxS+1
    br2 = if r1 >= maxS  then maxU else r1+maxS+1
-- ...

    gl1 = l1
    gr1 = if r2 < maxS+2 then 0    else r2-maxS-2
    gl2 = if l1 >= maxS  then maxU else l1+maxS+2
    gr2 = r2

    a = ((FlagsLattice BitTrue  BitFalse BitTrue  BitFalse),
        ( checkInterval (Interval (max l1 al1) (min r1 ar1))
        , checkInterval (Interval (max l2 al2) (min r2 ar2))))
    b = ((FlagsLattice BitTrue  BitFalse BitTrue  BitFalse),
        ( checkInterval (Interval (max l1 bl1) (min r1 br1))
        , checkInterval (Interval (max l2 bl2) (min r2 br2))))
-- ...

    g = ((FlagsLattice BitTrue  BitFalse BitTrue  BitFalse),
        ( checkInterval (Interval (max l1 gl1) (min r1 gr1))
        , checkInterval (Interval (max l2 gl2) (min r2 gr2))))

  in
    [
        (flgs,(extractJustContent l, extractJustContent r)) |
        (flgs,(l,r)) <- [a,b,c,d,e,f,g] , l /= Nothing, r /= Nothing
    ]

maxS :: Value
maxS = 2^(bitWidth-1)-1

maxU :: Value
maxU = (maxBound::Value)

checkInterval :: Interval -> Maybe Interval
checkInterval i@(Interval l r) = if l <= r then Just i else Nothing

extractJustContent :: Maybe a -> a
extractJustContent (Just x) = x
```

Listing 5: Excerpt of the code implementing *split*$_{\text{sub}}$ in an efficient manner

5.3 Bitflip Analysis

In this subsection we describe how our novel bitflip analysis is designed and implemented. It is largely built on top of the interval analysis described in the previous subsection. Conceptually, we want a result of our analysis to be a function from all potential bitflips that have occurred in a specific register at a specific program point to the relevant interval analysis result. We dub each pair of input and output of this function a *world* because of the intuition that they each refer to a distinct world where a bitflip happened somewhere. The lattice, L, of our analysis then, is the set of all these possible functions:

$$L = K \to V$$

The input or keys to these mappings is a tuple of the program point (or label) at which the bitflip occurred and the register in which it occurred. Additionally, we use ω to designate the key where no bitflip has occurred:

$$K = (\mathit{Label} \times \mathit{GeneralRegister}) \cup \{\omega\}$$

The output or values of the mappings is a tuple similar to the tuples used in the interval analysis, except to better track the consequences of individual bitflips, we annotate each of them with two sets of integers, describing which bits might have flipped. This part is parameterised to the bit width of the analysis, bw:

$$\begin{aligned} &V = \mathcal{P}(L_f \times \mathit{Vars} \times B) \\ &B = \mathcal{P}(\mathbb{N}_{bw-1})^2 \end{aligned}$$

The presence of an integer in the first set means it could be the bit that flipped from 0 to 1, while the second set describes the bits that could have flipped from 1 to 0. We refer to these types of bitflips as positive and negative respectively.

The following example of a single lattice element in the bitflip analysis should better illustrate what the set of bit indexes represents:

$$\begin{array}{ll} [& \\ \quad \omega & \mapsto \{((\bot,\bot,\bot,\bot),[r_1 \mapsto [42..42], r_2 \mapsto \bot, \ldots], (\emptyset,\emptyset)) \\ \quad (\texttt{lbl:1}, r_1) & \mapsto \{((\bot,\bot,\bot,\bot),[r_1 \mapsto [10..170], r_2 \mapsto \bot, \ldots], (\{0,2,4,6,7\},\{1,3,5\})) \\ \quad (\texttt{lbl:1}, r_2) & \mapsto \{\ldots\}, \\ \quad \ldots & \\ ] & \end{array}$$

The key $(\texttt{lbl:1}, r_1)$ describes that a bitflip happened in register r_1 just before the instruction at label `lbl:1` was executed. The set $\{0,2,4,6,7\}$ contains that the indices of the bits that could have flipped positively for the interval $[42..42]$ to extend its upper bound to 170 while $\{1,3,5\}$ are the indices of the bits that when flipped negative could lead to its lower bound being lowered to 10. Additionally, it should be noted that an analysis result technically includes a world for each combination of label and register, but the worlds which are not relevant (for example, if the instruction with label `lbl:1` does not use r_2, the third world in the example) simply contain the same set of tuples as ω. In the implementation of the analysis, this is slightly different, as we simply do not create the irrelevant worlds.

The *out* function is pretty much identical to that of the interval analysis, except of course it refers to the transfer functions of this analysis, and most transfer functions are parameterised with the label of the CFG node.

$$out(n) = \begin{cases} t_{\texttt{MOV}v}(r, v, in(n)) & \text{if } n = \texttt{MOV}_\chi\ r, v \\ t_{\texttt{MOV}r}(r_{dst}, r_{src}, label(n), in(n)) & \text{if } n = \texttt{MOV}_\chi\ r_{dst}, r_{src} \\ t_{\texttt{LDR}v}(r, in(n)) & \text{if } n = \texttt{LDR}_\chi\ r, v \\ t_{\texttt{LDR}r}(r_{dst}, r_{addr}, label(n), in(n)) & \text{if } n = \texttt{LDR}_\chi\ r_{dst}, r_{addr} \\ t_{\texttt{STR}v}(\{r\}, label(n), in(n)) & \text{if } n = \texttt{STR}_\chi\ r, v \\ t_{\texttt{STR}r}(\{r_x, r_y\}, label(n), in(n)) & \text{if } n = \texttt{STR}_\chi\ r_x, r_y \\ t_{\texttt{ADD}v}(r_{dst}, r_{src}, v, label(n), in(n)) & \text{if } n = \texttt{ADD}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{ADD}r}(r_{dst}, r_{src1}, r_{src2}, label(n), in(n)) & \text{if } n = \texttt{ADD}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{ADDS}v}(r_{dst}, r_{src}, v, label(n), in(n)) & \text{if } n = \texttt{ADDS}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{ADDS}r}(r_{dst}, r_{src1}, r_{src2}, label(n), in(n)) & \text{if } n = \texttt{ADDS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{SUB}v}(r_{dst}, r_{src}, v, label(n), in(n)) & \text{if } n = \texttt{SUB}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{SUB}r}(r_{dst}, r_{src1}, r_{src2}, label(n), in(n)) & \text{if } n = \texttt{SUB}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{SUBS}v}(r_{dst}, r_{src}, v, label(n), in(n)) & \text{if } n = \texttt{SUBS}_\chi\ r_{dst}, r_{src}, v \\ t_{\texttt{SUBS}r}(r_{dst}, r_{src1}, r_{src2}, label(n), in(n)) & \text{if } n = \texttt{SUBS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\ t_{\texttt{CMP}v-}(r, v, label(n), in(n)) & \text{if } n = \texttt{CMP}_\chi\ r, v \\ t_{\texttt{CMP}r-}(r_x, r_y, label(n), in(n)) & \text{if } n = \texttt{CMP}_\chi\ r_x, r_y \\ t_{\texttt{ASSERT}}(\chi, in(n)) & \text{if } n = \texttt{ASSERT}\ \chi \\ t_{\texttt{REFUTE}}(\chi, in(n)) & \text{if } n = \texttt{REFUTE}\ \chi \\ in(n) & \text{otherwise} \end{cases}$$

As introducing new worlds into the result can make the analysis results more difficult to comprehend, we only do so when strictly necessary. In practice this means that a bitflip cannot actually happen before all instructions, diverging from the definition of our fault model. The only functions that cannot introduce bitflips are `MOV` and `LDR`, but only when their second argument is a constant, and we would argue that any bitflip that were to happen right before one of those instructions can happen just before any other instruction that makes use of them them, with the same end result. We found which instruction should be able to introduce bitflips by looking at the instructions that Hansen et al. [7] refer to as *observable actions* and working our way backwards: if a store instruction can lead to observable actions then any instruction that can write a value to a register that can later be stored, is an instruction for which we must introduce a new world.

In the same style as with the interval analysis, we require a default state that contains some structure for our transfer functions to work as intended:

$$\lambda(k).\{((\bot, \bot, \bot, \bot), (\lambda(r).\bot), (\emptyset, \emptyset))\}$$

The transfer functions that do not introduce bitflips are similar to their interval analysis counterparts, except that they are modified to work on a different underlying structure, as can be seen in the definition of $t_{\texttt{LDR}v}$:

$$t_{\texttt{LDR}v} : \mathit{GeneralRegister} \times L \to L$$
$$t_{\texttt{LDR}v}(r, l) = \mathit{applyAll}(\lambda(X).\{(f, i[r \mapsto [0..MAX]], b) \mid (f, i, b) \in X\})(l)$$

$$\mathit{applyAll} : (V \to V) \to L \to L$$
$$\mathit{applyAll}(f)(l) = \lambda(k).f(l(k))$$

As the example in Section 1 shows, a bit flipping in a register that is used in a comparison can have quite serious consequences for the security of an entire program. Therefore the `CMP` instruction is of great interest to us and the analysis. Generally, the transfer function for `CMP`, $t_{\texttt{CMP}r-}$, looks similar to its counterpart in the interval analysis. The main differences are that it operates on a slightly different structure and that it is wrapped in a call to the function *flip*:

$$\begin{aligned}
&t_{\texttt{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\
&t_{\texttt{CMP}r-}(r_x, r_y, \mathit{lbl}, l) = \mathit{flip}(t'_{\texttt{CMP}r-}(r_x, r_y), \mathit{lbl}, \{r_x, r_y\}, l)
\end{aligned}$$

$$\begin{aligned}
&t'_{\texttt{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \to L \to L \\
&t'_{\texttt{CMP}r-}(r_x, r_y) = \mathit{applyAll}(\lambda(X).\{(f', i[r_x \mapsto s_x][r_y \mapsto s_y], b) \mid (\_, i, b) \in X, (f', s_x, s_y) \in \mathit{split}_{\texttt{sub}}(i(r_x), i(r_y))\})
\end{aligned}$$

flip handles the modelling of possible bitflips in multiple steps. The first step is the call to *expand*. For most worlds the function is simply the identity function, however for the keys with a label and registers that match the ones from the node *flip* is being called from, we are interested in widening any interval associated with the register. This is accomplished by the aptly named function *widenInterval*, which creates a new interval with a lower value equal to the smallest value that flipping a bit negatively can result in and vice versa for the greatest value. We present the definition of *widenInterval* as Haskell code in Listing 6 as we find it to be simpler to read.

$$\begin{aligned}
&\mathit{flip} : (L \to L) \times \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \to L \\
&\mathit{flip}(t, \mathit{lbl}, R, l) = \mathit{flip}'(\mathit{lbl}, R, l, t(\mathit{expand}(\mathit{lbl}, R, l)))
\end{aligned}$$

$$\begin{aligned}
&\mathit{expand} : \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \to L \\
&\mathit{expand}(\mathit{lbl}, R, l) = \lambda(\mathit{lbl}', r).\begin{cases} \mathit{expand}'(r, l(\omega)) & \text{if } r \in R \wedge \mathit{lbl} = \mathit{lbl}' \\ l(\mathit{lbl}', r) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathit{expand}' : \mathit{GeneralRegister} \times V \to V \\
&\mathit{expand}'(r, (f, i, b)) = (f, i[r \mapsto \mathit{widenInterval}(i(r))], b)
\end{aligned}$$

$$\begin{aligned}
&\mathit{widenInterval} : L_{i_{MAX}} \to L_{i_{MAX}} \\
&\mathit{widenInterval}(\bot) = \bot \\
&\mathit{widenInterval}([l..r]) = \ldots
\end{aligned}$$

```haskell
widenInterval :: Interval -> Interval
widenInterval (Interval min max) = Interval newMin newMax
  where
    msbIndex = (finiteBitSize max) - 1
    newMin = findNewMin min max (2^msbIndex)
    newMax = findNewMax min max (2^msbIndex)

    findNewMin :: Value -> Value -> Value -> Value
    findNewMin min max 0 = 0
    findNewMin min max powerOfTwo =
      if min >= powerOfTwo
        then min - powerOfTwo
        else if max >= powerOfTwo
          then 0
          else findNewMin min max (div powerOfTwo 2)

    findNewMax :: Value -> Value -> Value -> Value
    findNewMax min max 0 = max
    findNewMax min max powerOfTwo =
      if max < powerOfTwo
        then max + powerOfTwo
        else if min < powerOfTwo
          then (powerOfTwo * 2) - 1
          else powerOfTwo + findNewMax (min - powerOfTwo) (max - powerOfTwo) (div powerOfTwo 2)
```

Listing 6: Haskell implementation of the *widenInterval* function

If flip is called from a transfer function for an instruction which set the control flags, the new widened intervals must also be split according to the rules of the transfer function. This is done by a call to the t function which is passed to flip by the transfer function.

The next step is a call to *flip′*, which is passed the original input and the input after the intervals have been widened (and possibly split). For most keys the function simply returns the original input where nothing is changed. However, for the key where the label and register matches, the sets of bits that could have flipped are updated using the *possibleFlips* function. Again, we find it is easier to convey this function as Haskell code, and its implementation can be seen in Listing 7 with some helper functions shown in Listing 8

$$\begin{aligned}
&\mathit{flip}' : \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \times L \to L \\
&\mathit{flip}'(\mathit{lbl}, R, l_{pre}, l_{post}) = \lambda(k). \begin{cases} (f', i', \mathit{possibleFlips}(i(r), i'(r))) & \text{if } (\mathit{lbl}, r) \in k \wedge r \in R \\ l_{post}(k) & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\qquad\quad \text{where } (f, i, \_) \in l_{pre}(k), (f', i', \_) \in l_{post}(k)
\end{aligned}$$

$$\begin{aligned}
&\mathit{possibleFlips} : L_{i_{MAX}} \times L_{i_{MAX}} \to B \\
&\mathit{possibleFlips}(\bot, \_) = \emptyset \\
&\mathit{possibleFlips}(\_, \bot) = \emptyset \\
&\mathit{possibleFlips}([l_{pre}..r_{pre}], [l_{post}..r_{post}]) = \ldots
\end{aligned}$$

```haskell
possibleFlips :: Interval -> Interval -> ([Value], [Value])
possibleFlips intervalFrom intervalTo = (positiveFlips, negativeFlips)
  where
    msbIndex = bitWidth - 1
    positiveFlips = filter (canBitBeFlippedPos intervalFrom intervalTo) [0..msbIndex]
    negativeFlips = filter (canBitBeFlippedNeg intervalFrom intervalTo) [0..msbIndex]

    canBitBeFlippedPos :: Interval -> Interval -> Value -> Bool
    canBitBeFlippedPos from@(Interval origMin origMax) to@(Interval expMin expMax) bit =
      case (trimmedFrom, trimmedTo) of
        (Just tfrom, Just tto) ->
          case intervalIntersection tfrom $ intervalSubConst tto power of
            Nothing -> False -- No overlap
            _ -> True -- some overlap
        _ -> False -- either interval couldnt be flipped from/to
      where
        power = 2^bit
        trimmedFrom = trimIntervalToBitClear from bit
        trimmedTo   = trimIntervalToBitSet   to   bit

    canBitBeFlippedNeg :: Interval -> Interval -> Value -> Bool
    canBitBeFlippedNeg from@(Interval origMin origMax) to@(Interval expMin expMax) bit =
      case (trimmedFrom, trimmedTo) of
        (Just tfrom, Just tto) ->
          case intervalIntersection tfrom $ intervalAddConst tto power of
            Nothing -> False -- No overlap
            _ -> True -- some overlap
        _ -> False -- either interval couldnt be flipped from/to
      where
        power = 2^bit
        trimmedFrom = trimIntervalToBitSet   from bit
        trimmedTo   = trimIntervalToBitClear to   bit
```

Listing 7: Haskell implementation of the *possibleFlips* function

```haskell
trimIntervalToBitClear :: Interval -> Value -> Maybe Interval
trimIntervalToBitClear (Interval l r) bit =
  let ltrim = nextValueWithBitClear l bit
      rtrim = prevValueWithBitClear r bit
  in if ltrim >= l && rtrim <= r then maybeValidInterval (Interval ltrim rtrim) else Nothing

trimIntervalToBitSet :: Interval -> Value -> Maybe Interval
trimIntervalToBitSet (Interval l r) bit =
  let ltrim = nextValueWithBitSet l bit
      rtrim = prevValueWithBitSet r bit
  in if ltrim >= l && rtrim <= r then maybeValidInterval (Interval ltrim rtrim) else Nothing

nextValueWithBitSet :: Value -> Value -> Value
nextValueWithBitSet val bit =
  let power = 2^bit
  in if (val .&. power) == power then val else val + power - (val .&. (power-1))

nextValueWithBitClear :: Value -> Value -> Value
nextValueWithBitClear val bit =
  let power = 2^bit
  in if (val .&. power) == 0 then val else val + power - (val .&. (power-1))

prevValueWithBitSet :: Value -> Value -> Value
prevValueWithBitSet val bit =
  let power = 2^bit
  in if (val .&. power) == power then val else val - (val .&. (power-1)) -1

prevValueWithBitClear :: Value -> Value -> Value
prevValueWithBitClear val bit =
  let power = 2^bit
  in if (val .&. power) == 0 then val else val - (val .&. (power-1)) -1
```

Listing 8: Haskell implementation of helper functions for the *possibleFlips* function

As previously mentioned, we are interested in *flip* behaving differently depending on whether or not it is used by a transfer function that sets flags or not. It all comes down to *when* we want to calculate which bits could have flipped. One of the arguments to *flip* is a transfer function, t_1, but flip is also called inside a transfer function, t_2. We can describe this as $t_2(flip(t_1, l))$, where the important distinction is that t_1 is applied in between the interval widening and the calculation of which specific bits have flipped. For an operation that updates a register we want our transfer, t to happen after the bitflips have been calculated, i.e. $t(flip(id, l))$, where id is the identity function. For an operation that sets flags, but does not update a register we want our entire transfer, t to happen in between calculations, i.e. $id(flip(t, l))$ For operations that both set flags and updates a register we apply the transfer in two passes, i.e. $t_2(flip(t_1, l))$.

We have already shown an example of the second case with the the $t_{\texttt{CMP}r-}$ function. The following functions $t_{\mathsf{SUB}r}$ and $t_{\mathsf{SUBS}r}$ are examples of the first and third type respectively:

$$\begin{aligned}
&t_{\mathsf{SUB}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L\\
&t_{\mathsf{SUB}r}(r_{\mathit{dst}}, r_{\mathit{src1}}, r_{\mathit{src2}}, \mathit{lbl}, l) = \mathit{applyAll}(\\
&\qquad \lambda(X).\{(f, i[r_{\mathit{dst}} \mapsto \mathit{sub}_I(i(r_{\mathit{src1}}), i(r_{\mathit{src2}}))], b) \mid (f, i, b) \in X\})(\\
&\qquad \mathit{flip}(t_{\mathit{id}}, \mathit{lbl}, \{r_{\mathit{src1}}, r_{\mathit{src2}}\}, l))
\end{aligned}$$

$$\begin{aligned}
&t_{\mathit{id}} : L \to L\\
&t_{\mathit{id}}(l) = l
\end{aligned}$$

$$\begin{aligned}
&t_{\mathsf{SUBS}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L\\
&t_{\mathsf{SUBS}r}(r_{\mathit{dst}}, r_{\mathit{src1}}, r_{\mathit{src2}}, \mathit{lbl}, l) = \mathit{applyAll}(\\
&\qquad \lambda(X).\{(f, i[r_{\mathit{dst}} \mapsto \mathit{sub}_I(i(r_{\mathit{src1}}), i(r_{\mathit{src2}}))], b) \mid (f, i, b) \in X\})(\\
&\qquad \mathit{flip}(t'_{\mathsf{CMP}r-}(r_{\mathit{src1}}, r_{\mathit{src2}}), \mathit{lbl}, \{r_{\mathit{src1}}, r_{\mathit{src2}}\}, l))
\end{aligned}$$

While we are unfortunately unable to prove the monotonicity of the transfer functions we use in the analysis, we have yet to see example program for which our analysis does not terminate. In the next section we present several examples of output from both our analyses on what we deem to be interesting programs.

6 Results

In this section we discuss the results of running our analyses on a few programs to show how they actually perform. All the analysis results we present are run on an 8 bit version of TinyARM to make the output easier to read. Additionally, when we present the results of our analysis, we do so only for a single program point, showing only valid worlds and registers that are not ⊥. We also present a few additional features implemented in our analysis program, such as different output modes and the support for extending it with multiple analyses.

6.1 Sensor Values

The program shown in Listing 9 is our attempt at modelling a small part of some system that reads a sensor value or similar from some memory mapped input and performs several checks on it to make sure it is within some predefined bounds before performing some (possibly) dangerous operation if the value passes all checks.

```
    ldr r1, #21
    cmp r1, #49
    b_ls error
    cmp r1, #120
    b_hi error
    cmp r1, #100
    b_eq error
safe:
    out #0 ; result should be [50..99] [101..120]
    b end
error:
    out #1
end:
```

Listing 9: A simple program that performs several checks on a value loaded from memory

The first world ω shows which values r_1 can have at the label `safe`. This result is equivalent to what our interval analysis outputs. The three remaining worlds are all ones where a bitflip has occurred, with the intervals showing how they would look after such a bitflip. Depending on what happens once the `safe` label is reached there are two types of problems with the program. If we wish to make use of the value, but first make sure it is in the intervals [50..99] and [101..120], then the two final worlds show us that we cannot trust that the value is in these intervals, as it can in fact be any value in the interval [0..248] except 100. If we do not use the value directly, but simply perform checks on it to reason about the state of the system as measured by the sensor, all three worlds are problematic. For example, the first tuple in the world $(\texttt{lbl:2}, r_1)$ shows bit 6 could have flipped positive to add 64 to some value that was originally lower than 50, and example is $35 + 64 = 99$. Reasoning about this result is not entirely simple and requires intimate knowledge of the program itself, but given that the analysis is a may analysis, this is not really how it is used most effectively. Instead, a result where no bitflip can get us to a certain state actually gives us a guarantee about the safety of the program.

$$
\begin{array}{l}
[\\
\quad \omega \mapsto\\
\qquad \{(([1..1],[0..0],[0..0],[0..0]),[r_1 \mapsto [50..99]],(\{\},\{\}))\\
\qquad ,(([0..0],[0..0],[1..1],[0..0]),[r_1 \mapsto [101..120]],(\{\},\{\}))\},\\
\quad (\texttt{lbl:2}, r_1) \mapsto\\
\qquad \{(([1..1],[0..0],[0..0],[0..0]),[r_1 \mapsto [50..99]],(\{0,1,2,3,4,5,6\},\{0,1,2,3,4,5,6,7\}))\\
\qquad ,(([0..0],[0..0],[1..1],[0..0]),[r_1 \mapsto [101..120]],(\{0,1,2,3,4,5,6\},\{0,1,2,3,4,5,6,7\}))\},\\
\quad (\texttt{lbl:4}, r_1) \mapsto\\
\qquad \{(([1..1],[0..0],[0..0],[0..0]),[r_1 \mapsto [0..99]],(\{0,1,2,3,4,5,6\},\{0,1,2,3,4,5,6,7\}))\\
\qquad ,(([0..0],[0..0],[1..1],[0..0]),[r_1 \mapsto [101..119]],(\{0,1,2,3,4,5,6\},\{0,1,2,3,4,5,6,7\}))\\
\qquad ,(([0..0],[0..0],[1..1],[0..0]),[r_1 \mapsto [120..120]],(\{3,4,5,6\},\{0,1,2,7\}))\},\\
\quad (\texttt{lbl:6}, r_1) \mapsto\\
\qquad \{(([1..1],[0..0],[0..0],[0..0]),[r_1 \mapsto [0..99]],(\{0,1,2,3,4,5\},\{0,1,2,3,4,5,6\}))\\
\qquad ,(([0..0],[0..0],[1..1],[0..0]),[r_1 \mapsto [101..127]],(\{0,1,2,3,4,5,6\},\{0,1,2,3,4\}))\\
\qquad ,(([0..0],[0..0],[1..1],[1..1]),[r_1 \mapsto [128..227]],(\{7\},\{\}))\\
\qquad ,(([1..1],[0..0],[1..1],[0..0]),[r_1 \mapsto [228..248]],(\{7\},\{\}))\},\\
]
\end{array}
$$

6.2 Robust Assert

As previously discussed, the robust assert gadget presented by Hansen et al. [7] is interesting to us, as we would expect our analysis to corroborate their findings, that whether a bitflip occurs or not, the `success` label can only be reached if the contents of the registers r_2 and r_3 are equal at the beginning of the gadget. Running our analysis on the two TinyARM programs seen in Figure 9 gives the following results at the `success` label:

$$
\begin{array}{l}
[\\
\quad \begin{array}{ll}
\omega & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_1 \mapsto [0..0], r_2 \mapsto [42..42], r_3 \mapsto [42..42]],(\emptyset,\emptyset))\} \\
(\texttt{lbl:3}, r_2) & \mapsto \emptyset \\
(\texttt{lbl:3}, r_3) & \mapsto \emptyset \\
(\texttt{lbl:5}, r_1) & \mapsto \emptyset
\end{array} \\
] \\
[\\
\quad \begin{array}{ll}
\omega & \mapsto \emptyset \\
(\texttt{lbl:3}, r_2) & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_1 \mapsto [0..0], r_2 \mapsto [10..10], r_3 \mapsto [10..10]],(\emptyset,\{5\}))\} \\
(\texttt{lbl:3}, r_3) & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_1 \mapsto [0..0], r_2 \mapsto [42..42], r_3 \mapsto [42..42]],(\{5\},\emptyset))\} \\
(\texttt{lbl:5}, r_1) & \mapsto \emptyset
\end{array} \\
]
\end{array}
$$

The first result clearly shows that the only possible way to reach the success label is if no bitflips happen, as any bitflips before the gadget will make the two values different from each other and a bitflip inside the gadget will change the difference between the values to something other than 0. The second result is slightly more interesting, as it actually shows there are two possible worlds where a bitflip can bring us to the success label even though the two values differ to begin with. The values do have to differ by some power of two for a single bitflip to make them equal though, in this example 32, which is why bit 5 (0-indexed) is the one that can flip positively in r_3 or negatively in r_2 to make the registers contain the same value. This is still in line with the robust assert gadget's purpose, as the bitflips were only possible outside the gadget itself, as the empty set in the fourth world shows.

```
1    mov r2, #42
2    mov r3, #42
3    subs r1, r2, r3    ; start gadget
4    b_ne fail
5    cmp r1, #0
6    b_ne fail          ; end gadget
7  success:
8    out #0
9    b end
10 fail:
11   out #1
12 end:
```

```
1    mov r2, #42
2    mov r3, #10
3    subs r1, r2, r3    ; start gadget
4    b_ne fail
5    cmp r1, #0
6    b_ne fail          ; end gadget
7  success:
8    out #0
9    b end
10 fail:
11   out #1
12 end:
```

Figure 9: TinyARM programs using a robust assert gadget where $r_2 = r_3$ and $r_2 \neq r_3$

However, running the programs shown in Figure 10, which are similar but only make use of a single comparison gives almost identical results:

$$
\begin{array}{ll}
[& \\
\quad \omega & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_2 \mapsto [42..42], r_3 \mapsto [42..42]],(\emptyset,\emptyset))\} \\
\quad (\texttt{lbl:3}, r_2) & \mapsto \emptyset \\
\quad (\texttt{lbl:3}, r_3) & \mapsto \emptyset \\
] & \\
[& \\
\quad \omega & \mapsto \emptyset \\
\quad (\texttt{lbl:3}, r_2) & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_2 \mapsto [10..10], r_3 \mapsto [10..10]],(\emptyset,\{5\}))\} \\
\quad (\texttt{lbl:3}, r_3) & \mapsto \{(([0..0],[1..1],[1..1],[0..0]),[r_2 \mapsto [42..42], r_3 \mapsto [42..42]],(\{5\},\emptyset))\} \\
] &
\end{array}
$$

This would seem to indicate that the extra comparison in the robust assert gadget does not actually add any robustness, which is at least partly correct. The problem lies in the fact that the gadget is supposed to protect against a more aggressive fault model where it is also possible for individual flags to flip, which means that the result of a single comparison cannot be trusted. Expanding our analysis to also support this fault model is entirely possible as there is nothing stopping the *flip* function from accessing and changing the values stored in the flags, but due to time constraints it is something we leave for future work.

```
  mov r2, #42
  mov r3, #42
  cmp r2, r2,
  b_ne fail
success:
  out #0
  b end
fail:
  out #1
end:
```

```
  mov r2, #42
  mov r3, #10
  cmp r2, r2,
  b_ne fail
success:
  out #0
  b end
fail:
  out #1
end:
```

Figure 10: TinyARM programs using a single compare where $r_2 = r_3$ and $r_2 \neq r_3$

6.3 Code Duplication

If we consider the code in Figure 10 as a crude modelling of the kind of logic that could be present in a system that checks if a code entered on a keypad matches a stored one. Focusing on the version where the values are different, we can then imagine how this is equivalent to someone entering the wrong passcode. As already seen, it is possible for someone to enter a wrong code and still gain access (reach the label `success`) if a bit flips in either the entered code or the stored correct code. By duplicating each value and any operation on them and keeping the two in a separate set of registers, we are relatively close to adhering to the scheme presented by Oh et al. in [12]. Before making use of the two independent chain of operations we must compare their results. A version of the code that utilises this approach can be seen in Listing 10

```
1      mov r1,  #42
2      mov r11, #42
3      mov r2,  #10
4      mov r12, #10
5      subs r3, r1, r2
6      b_ne fail
7      subs r4, r11, r12
8      b_ne fail
9      cmp r3, r4
10     b_ne fail
11 success:
12     out #0
13     b end
14 fail:
15     out #1
16 end:
```

Listing 10: Using code duplication to avoid reaching `success` even with bitflips

As expected, the result of the analysis at the `success` label shows that there is no possible way to end up there, regardless of bitflips:

$$
\begin{array}{ll}
[& \\
\quad \omega & \mapsto \emptyset \\
\quad (\texttt{lbl:5}, r_1) & \mapsto \emptyset \\
\quad (\texttt{lbl:5}, r_2) & \mapsto \emptyset \\
\quad (\texttt{lbl:7}, r_{11}) & \mapsto \emptyset \\
\quad (\texttt{lbl:7}, r_{12}) & \mapsto \emptyset \\
\quad (\texttt{lbl:9}, r_3) & \mapsto \emptyset \\
\quad (\texttt{lbl:9}, r_4) & \mapsto \emptyset \\
] &
\end{array}
$$

Obviously, in the real world the entered passcode would be loaded into registers using an `ldr` instruction and we would not necessarily know that the password is incorrect. Our analysis *can* be used to show that we are still protected against bitflips, but it requires careful reading of the results, which initially seem to indicate that there are bitflips that cause us to reach `success` without entering the correct passcode:

$$
\begin{array}{l}
[\\
\quad \omega \mapsto \{(([0..0],[1..1],[1..1],[0..0]), \\
[r_1 \mapsto [42..42], r_2 \mapsto [42..42], r_3 \mapsto [0..0], r_{11} \mapsto [42..42], r_{12} \mapsto [42..42], r_4 \mapsto [0..0]], (\emptyset, \emptyset))\}, \\
\quad (\texttt{lbl:5}, r_1) \mapsto \{(([0..0],[1..1],[1..1],[0..0]), \\
[r_1 \mapsto [10..170], r_2 \mapsto [10..170], r_3 \mapsto [0..0], r_{11} \mapsto [42..42], r_{12} \mapsto [42..42], r_4 \mapsto [0..0]], (\{0,2,4,6,7\}, \{1,3,5\}))\}, \\
\quad (\texttt{lbl:5}, r_2) \mapsto \{(([0..0],[1..1],[1..1],[0..0]), \\
[r_1 \mapsto [42..42], r_2 \mapsto [42..42], r_3 \mapsto [0..0], r_{11} \mapsto [42..42], r_{12} \mapsto [42..42], r_4 \mapsto [0..0]], (\{1,3,5\}, \{0,2,4,6,7\}))\}, \\
\quad (\texttt{lbl:7}, r_1 1) \mapsto \{(([0..0],[1..1],[1..1],[0..0]), \\
[r_1 \mapsto [42..42], r_2 \mapsto [42..42], r_3 \mapsto [0..0], r_{11} \mapsto [10..170], r_{12} \mapsto [10..170], r_4 \mapsto [0..0]], (\{0,2,4,6,7\}, \{1,3,5\}))\}, \\
\quad (\texttt{lbl:7}, r_1 2) \mapsto \{(([0..0],[1..1],[1..1],[0..0]), \\
[r_1 \mapsto [42..42], r_2 \mapsto [42..42], r_3 \mapsto [0..0], r_{11} \mapsto [42..42], r_{12} \mapsto [42..42], r_4 \mapsto [0..0]], (\{1,3,5\}, \{0,2,4,6,7\}))\}, \\
\quad (\texttt{lbl:9}, r_3) \mapsto \emptyset, \\
\quad (\texttt{lbl:9}, r_4) \mapsto \emptyset \\
]
\end{array}
$$

For example, looking at the world where a bit flipped in r_1 in the first comparison between r_1 and r_2, it looks as if that flip made it possible for the correct code (42) to be flipped to another value in the interval [10..170], which could have made it match the entered passcode even if it was correct. But given that we know that two the subsequent loads from the same memory address are the same, we notice that the two registers r_2 and r_{12} are no longer equal. While the interval of r_2 does contain the value r_{12}, there is no way for r_2 to have been widened to be different from r_{12} if the bitflip happened in r_1. Our analysis does not support putting this constraint on the results from `ldr`, but it is possible to model it using global value numbering or a similar technique. We leave the implementation of such a feature to future work.

6.4 Additional Features

While our work and certainly this report has mainly focused on the two analyses we have created, a substantial amount of effort has gone in to the development of the tool we use to run our analyses through. The tool has been created with expandability in mind, such that adding additional types of data flow analyses to it is simply a matter of defining lattices and transfer functions for them. Obviously, the tool contains a parser for the TinyARM language, but a simplistic interpreter for the language is also included, which can be used to check if programs actually behave as expected. Figure 11 contains a simple program that calculates and outputs the sequence of Fibonacci numbers stopping when the resulting numbers no longer fit in a register. The figure also contains the output our interpreter generates for the same program when run with a bit width of 8. The final state of the heap, registers and control flags are also printed by the interpreter.

```
  mov r1, #0
  mov r2, #1
  out r1
  out r2
loop:
  adds r3, r1, r2
  mov r1, r2
  mov r2, r3
  out_cc r3
  b_cc loop
```

```
---------------
-- Heap      --
---------------

---------------
-- Registers --
---------------
PC = 9
r1 = 233
r2 = 121
r3 = 121

---------------
-- Flags     --
---------------
NZCV
0010
```

```
---------------
-- Output    --
---------------
0
1
1
2
3
5
8
13
21
34
55
89
144
233
HALT: end of program
```

Figure 11: TinyARM implementation of a program that prints numbers from the Fibonacci sequence and its output when run on 8 bits

The tool also support outputting CFGs of programs. The CFG of the robust assert gadget program from Figure 9 (right) can be seen in Figure 12. The results of analyses can be outputted in two ways: a pretty printed text format or on an annotated CFG.

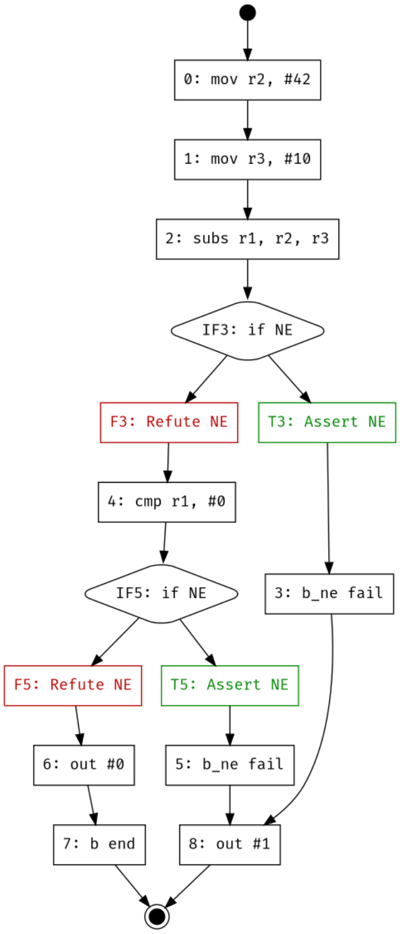

Figure 12: CFG for the robust assert gadget

6.5 Shortcomings

We have already discussed a multitude of missing features in our final bitflip analysis, but in this subsection we cover a few more. Arguably, our most prominent shortcoming is that we lack any proofs that the transfer functions in the bitflip analysis are in fact monotonic, and in quite a few cases even any informal arguments as to why we expect the functions to behave nicely. This is simply the result of us prioritising the implementation of what we deem an interesting analysis over proving that it always terminates.

Running our analysis on a program similar to the PAM example would tell us that there are indeed values for `uid` that could be flipped to be 0, but it does not directly tell us which values those are. Clearly solving that problem is outside the scope of this report, but it does make for some interesting future work, perhaps by using an SMT based approach to finding these values and using the result of our current analysis to define some constraints to help solve the problem faster.

Our analysis does not accurately model bitflips happening at different iterations of a loop, as the worlds created in a loop are simply overwritten on the next iteration. A simple way of handling this, would be to unroll all loops for which we can calculate the number of iterations. Alternatively, we could join the worlds until we reach $\top$ on the relevant registers and both sets of bit indices are full, at which point any additional analysis is going to result in no changes.

An interesting but rather different fault model for the analysis to support, would be one that could skip a single instruction. Research shows that it is possible to deliberately skip practically any number of instructions on some architectures [13] and protection schemes have been proposed that can mitigate the effect of such an attack, at least for single instruction skips [5].

Finally, it is worth mentioning that while TinyARM does support calculated branch targets, our analysis does not. Implementing it could possibly be done rather simply by running the entire analysis multiple times and using the resulting intervals for the branch target to create an ever larger CFG where the relevant nodes have edges to several nodes in the program. Neither the efficacy or efficiency of such a solution is apparent, and again must be left for future work.

7 Conclusion

In this report we have presented our definition and implementation of two data flow analyses for the assembly language TinyARM. The first is an interval analysis that models the behaviour of control flags. The second, a novel analysis for showing how a single bit flipping during program execution can lead to otherwise unreachable states, is our main contribution. In order to properly present the definition of these we have covered relevant background theory of data flow analyses and lattices. This has also allowed us to define a monotone framework in a manner that suits how we define our analyses. Effort has been put into the implementation of the analyses such that they are able to perform efficiently on larger bit width versions of TinyARM without appreciable slowdown. The implementation also supports interpreting TinyARM programs and producing neatly formatted output such as CFGs. We have presented the result of running our bitflip analysis on multiple different programs to demonstrate the format and usefulness of its output, while also showing that it agrees with previously published schemes for protection against bitflips. Unfortunately, we fail to provide sufficient proofs for the monotonicity of most of the transfer functions and find that there are several improvements that could be made to the analysis to obtain more precision.

References

[1] Anton Christensen and Henrik Herbst Sørensen. Review of methods for handling bitflips using formal fault models, January 2020.

[2] Cody Maloney. auth.c - PAM authorization code, version 2.35, January 2020. URL: https://github.com/karelzak/util-linux/blob/master/login-utils/auth.c.

[3] J. G. Tront, J. R. Armstrong, and J. V. Oak. Software techniques for detecting single-event upsets in satellite computers. *IEEE Transactions on Nuclear Science*, 32(6):4225–4228, December 1985. ISSN: 1558-1578. DOI: 10.1109/TNS.1985.4334099.

[4] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, March 2005. DOI: 10.1109/CGO.2005.34.

[5] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, September 2014. ISSN: 2190-8516. DOI: 10.1007/s13389-014-0077-7. URL: https://doi.org/10.1007/s13389-014-0077-7.

[6] Frances Perry, Lester Mackey, George A. Reis, Jay Ligatti, David I. August, and David Walker. Fault-tolerant typed assembly language. *SIGPLAN Not.*, 42(6):42–53, June 2007. ISSN: 0362-1340. DOI: 10.1145/1273442.1250741.

[7] René Rydhof Hansen, Kim Guldstrand Larsen, Mads Chr. Olesen, and Erik Ramsgaard Wognsen. Formal modelling and analysis of bitflips in arm assembly code. *Information Systems Frontiers*, 18(5):909–925, October 2016.

[8] Gary Kildall. Global expression optimization during compilation, 1972. URL: http://search.proquest.com/docview/302615627/.

[9] Anders Møller and Michael I. Schwartzbach. Static program analysis, December 2019.

[10] B. A. Davey. *Introduction to lattices and order.* Cambridge University Press, New York, 2. ed. Edition, 2002. ISBN: 0521784514.

[11] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[12] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002. DOI: 10.1109/24.994913.

[13] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. *Experimental analysis of the laser-induced instruction skip fault model.* In November 2019, pages 221–237. ISBN: 978-3-030-35054-3. DOI: 10.1007/978-3-030-35055-0_14.

Appendices

A The TinyARM Language

A.1 Syntax and Semantics

We define the set of values as the set of integers that can be encoded as 32-bit wide binary numbers:

$$\mathbf{Val} = \mathbb{B}_{32}$$

where $\mathbb{B} = \{0, 1\}$ and $\mathbb{B}_n = [0..n-1] \rightarrow \mathbb{B}$, with 0 referring to the Least Significant Bit (LSB). Note that going forward, sets of functions are marked in bold. Additionally, for a binary digit, $b \in \mathbb{B}$, we define $\bar{b}$ to negate the bit. That is:

$$\bar{b} = 1 - b$$

Addresses of locations in the heap have the same size as values::

$$\mathbf{Addr} = \mathbf{Val}$$

We define 14 registers: 13 general purpose and the Program Counter (PC):

$$\begin{aligned}
&\text{GeneralRegister} = \{r_0, r_1, ..., r_{12}\} \\
&\text{ControlRegister} = \{r_{pc}\} \\
&\text{Register} = \text{GeneralRegister} \cup \text{ControlRegister}
\end{aligned}$$

As all registers hold values we define mappings from Register to **Val**:

$$\begin{aligned}
&\mathbf{GeneralRegisters} = \text{GeneralRegister} \rightarrow \mathbf{Val} \\
&\mathbf{ControlRegisters} = \text{ControlRegister} \rightarrow \mathbf{Val} \\
&\mathbf{Registers} = \text{Register} \rightarrow \mathbf{Val}
\end{aligned}$$

In addition to registers, the ARM architecture also makes use of four control flags, *Negative*, *Zero*, *Carry* and *Overflow*. These can be set by certain instructions and their contents used to determine if a specific condition holds. Each flag contains a single binary bit:

$$\begin{aligned}
&\text{Flag} = \{f_N, f_Z, f_C, f_V\} \\
&\mathbf{Flags} = \text{Flag} \rightarrow \mathbb{B}
\end{aligned}$$

In TinyARM conditional execution is accomplished by annotating individual instructions with a condition code, while unconditional execution is performed using the condition code `AL`. In contrast to regular ARM, we have chosen to include a condition code for unconditionally skipping an instruction:

$$\text{ConditionCode} = \{\texttt{EQ}, \texttt{NE}, \texttt{CS}, \texttt{CC}, \texttt{MI}, \texttt{PL}, \texttt{VS}, \texttt{VC}, \texttt{HI}, \texttt{LS}, \texttt{GE}, \texttt{LT}, \texttt{GT}, \texttt{LE}, \texttt{AL}, \texttt{NV}\}$$

The condition codes are used in combination with **Flags** in the function *cond* to determine whether or not to execute an instruction:

$$\begin{aligned}
&\mathit{cond}\colon \text{ConditionCode} \times \mathbf{Flags} \to \{\text{true}, \text{false}\}\\
&\mathit{cond}(\chi, F) = \begin{cases}
F(f_Z) = 1 & \text{if } \chi = \texttt{EQ}\\
F(f_Z) = 0 & \text{if } \chi = \texttt{NE}\\
F(f_C) = 1 & \text{if } \chi = \texttt{CS}\\
F(f_C) = 0 & \text{if } \chi = \texttt{CC}\\
F(f_N) = 1 & \text{if } \chi = \texttt{MI}\\
F(f_N) = 0 & \text{if } \chi = \texttt{PL}\\
F(f_V) = 1 & \text{if } \chi = \texttt{VS}\\
F(f_V) = 0 & \text{if } \chi = \texttt{VC}\\
F(f_C) = 1 \wedge F(f_Z) = 0 & \text{if } \chi = \texttt{HI}\\
F(f_C) = 0 \vee F(f_Z) = 1 & \text{if } \chi = \texttt{LS}\\
F(f_N) = F(f_V) & \text{if } \chi = \texttt{GE}\\
F(f_N) \neq F(f_V) & \text{if } \chi = \texttt{LT}\\
F(f_Z) = 0 \wedge F(f_N) = F(f_V) & \text{if } \chi = \texttt{GT}\\
F(f_Z) = 1 \vee F(f_N) \neq F(f_V) & \text{if } \chi = \texttt{LE}\\
\text{true} & \text{if } \chi = \texttt{AL}\\
\text{false} & \text{if } \chi = \texttt{NV}
\end{cases}
\end{aligned}$$

Instructions for which the condition code evaluates to false are caught by the semantic rule [nop], which simply increments the program counter.

We define functions for addition and subtraction and make use of the fact that our binary values allow us to extract individual bits.

We define the two arithmetic operators in TinyARM, addition and subtraction:

$$\text{Operator} = \{\texttt{ADD}, \texttt{SUB}\}$$

For addition we define the function add_{bin} such that it models binary addition, by recursively adding pairs of bits. It also requires the use of the helper function carry_{bin}, which returns the carry bits of the addition of its two inputs.

$$\begin{aligned}
&\mathit{carry}_{bin}\colon \mathbb{B}_n \times \mathbb{B}_n \times \{0,1\} \to \mathbb{B}_{n+1}\\
&\mathit{carry}_{bin}(v_1, v_2, c_{in})(n) = \begin{cases}
c_{in} & \text{if } n = 0\\
1 & \text{if } v_1(n-1) + v_2(n-1) + \mathit{carry}_{bin}(v_1, v_2, c_{in})(n-1) > 1\\
0 & \text{otherwise}
\end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathit{add}_{bin}\colon \mathbb{B}_n \times \mathbb{B}_n \times \{0,1\} \to \mathbb{B}_n\\
&\mathit{add}_{bin}(v_1, v_2, c_{in})(n) = v_1(n) + v_2(n) + \mathit{carry}_{bin}(v_1, v_2, c_{in})(n) \mod 2
\end{aligned}$$

Subtraction is implemented with the function sub_{bin}, which exploits the fact that subtracting two numbers is equivalent to negating the second number before adding it to the first. If signed integers are represented using two's complement, negating a number can simply be accomplished by inverting each bit (using the function inv_{bin}) and adding one to the result. Both add_{bin} and carry_{bin} take a third argument, which allows this addition of one.

$$\begin{aligned}
&\mathit{inv}_{bin} \colon \mathbb{B}_n \to \mathbb{B}_n \\
&\mathit{inv}_{bin}(v)(n) = \overline{v(n)}
\end{aligned}$$

$$\begin{aligned}
&\mathit{sub}_{bin} \colon \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n \\
&\mathit{sub}_{bin}(v_1, v_2)(n) = \mathit{add}_{bin}(v_1, \mathit{inv}_{bin}(v_2), 1)
\end{aligned}$$

We can now give a definition of the $\mathit{flags}_{\mathsf{ADD}}$ function:

$$\begin{aligned}
&\mathit{flags}_{\mathsf{ADD}} \colon \mathbf{Val} \times \mathbf{Val} \to \mathbf{Flags} \\
&\mathit{flags}_{\mathsf{ADD}}(v_1, v_2)(f_N) = \mathit{add}_{bin}(v_1, v_2, 0)(31)
\end{aligned}$$

$$\mathit{flags}_{\mathsf{ADD}}(v_1, v_2)(f_Z) = \begin{cases} 1 & \text{if } \forall n \in [0..31]\big(\mathit{add}_{bin}(v_1, v_2, 0)(n) = 0\big) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\mathsf{ADD}}(v_1, v_2)(f_C) = \mathit{carry}_{bin}(v_1, v_2, 0)(32)$$

$$\mathit{flags}_{\mathsf{ADD}}(v_1, v_2)(f_V) = \begin{cases} 1 & \text{if } v_1(31) = 0 \land v_2(31) = 0 \land \mathit{bin}_{add}(v_1, v_2, 0)(31) = 1 \\ 1 & \text{if } v_1(31) = 1 \land v_2(31) = 1 \land \mathit{bin}_{add}(v_1, v_2, 0)(31) = 0 \\ 0 & \text{otherwise} \end{cases}$$

And the $\mathit{flags}_{\mathsf{SUB}}$ function:

$$\begin{aligned}
&\mathit{flags}_{\mathsf{SUB}} \colon \mathbf{Val} \times \mathbf{Val} \to \mathbf{Flags} \\
&\mathit{flags}_{\mathsf{SUB}}(v_1, v_2)(f_N) = \mathit{sub}_{bin}(v_1, v_2)(31)
\end{aligned}$$

$$\mathit{flags}_{\mathsf{SUB}}(v_1, v_2)(f_Z) = \begin{cases} 1 & \text{if } \forall j \in [0, 1, \dots, 31] \quad \mathit{add}_{sub}(v_1, v_2)(j) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\mathsf{SUB}}(v_1, v_2)(f_C) = \mathit{carry}_{bin}(v_1, \mathit{inv}_{bin}(v_2), 1)(32)$$

$$\mathit{flags}_{\mathsf{SUB}}(v_1, v_2)(f_V) = \begin{cases} 1 & \text{if } v_1(31) = 0 \land v_2(31) = 1 \land \mathit{bin}_{sub}(v_1, v_2)(31) = 1 \\ 1 & \text{if } v_1(31) = 1 \land v_2(31) = 0 \land \mathit{bin}_{sub}(v_1, v_2)(31) = 0 \\ 0 & \text{otherwise} \end{cases}$$

We name the set of all instructions in TinyARM *Instr* and define a member of this set with the following grammar:

$$
\begin{array}{rlll}
\textit{instr} ::= & \mathtt{MOV}_\chi\ x, v & \text{store value } v \text{ in } x \\
\mid & \mathtt{MOV}_\chi\ x, y & \text{store value in } y \text{ in } x \\
\mid & \mathtt{ADD}_\chi\ x, y, z & \text{add value in } y \text{ to value in } z \text{ and store result in } x \\
\mid & \mathtt{ADD}_\chi\ x, y, v & \text{add value in } y \text{ to value } v \text{ and store result in } x \\
\mid & \mathtt{ADDS}_\chi\ x, y, z & \text{same as } \mathtt{ADD}\text{, but also set flags} \\
\mid & \mathtt{ADDS}_\chi\ x, y, v & \text{same as } \mathtt{ADD}\text{, but also set flags} \\
\mid & \mathtt{SUB}_\chi\ x, y, z & \text{subtract value in } z \text{ from value in } y \text{ and store result in } x \\
\mid & \mathtt{SUB}_\chi\ x, y, v & \text{subtract value } v \text{ from value in } y \text{ and store result in } x \\
\mid & \mathtt{SUBS}_\chi\ x, y, z & \text{same as } \mathtt{SUB}\text{, but also set flags} \\
\mid & \mathtt{SUBS}_\chi\ x, y, v & \text{same as } \mathtt{SUB}\text{, but also set flags} \\
\mid & \mathtt{CMP}_\chi\ x, y & \text{compare value in } x \text{ with value in } y \text{ and set flags} \\
\mid & \mathtt{CMP}_\chi\ x, v & \text{compare value in } x \text{ with value } v \text{ and set flags} \\
\mid & \mathtt{LDR}_\chi\ x, a & \text{store in } x \text{ the value at heap address } a \\
\mid & \mathtt{LDR}_\chi\ x, y & \text{store in } x \text{ the value at heap address in } y \\
\mid & \mathtt{STR}_\chi\ x, a & \text{store value in } x \text{ at heap address } a \\
\mid & \mathtt{STR}_\chi\ x, y & \text{store value in } x \text{ at heap address in } y \\
\mid & \mathtt{B}_\chi\ a & \text{store in } r_{\mathtt{PC}} \text{ the address } a \\
\mid & \mathtt{B}_\chi\ x & \text{store in } r_{\mathtt{PC}} \text{ the address in } x \\
\mid & \mathtt{NOP}_\chi & \text{increment program counter}
\end{array}
$$

where $\chi \in$ ConditionCode, $x, y, z \in$ GeneralRegister, $v \in \mathbf{Val}$ and $a \in \mathbf{Addr}$.

We can formalise a program as a mapping of addresses to instructions and heap memory as a mapping of addresses to values:

$$\mathbf{Program} = \mathbf{Addr} \to \textit{Instr}$$
$$\mathbf{Heap} = \mathbf{Addr} \to \mathbf{Val}$$

We formalise the configurations used in the structural operational semantics:

$$\text{Conf} = \mathbf{Program} \times \mathbf{Heap} \times \mathbf{Registers} \times \mathbf{Flags}$$

With $R = \mathbf{Registers}$ and for $x \in$ Register and $n \in \mathbb{Z}$ we define

$$R_{x+z} = R[x \to R(x) + n]$$

primarily, so we can write $R_{r_{pc}+1}$ to increment the program counter by one.

With this we can define the semantics of the language:

$$[\mathrm{mov}_{\mathrm{val}}] \qquad \frac{P(R(r_{pc})) = \mathsf{MOV}_\chi\ x, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto v], F\rangle}$$

$$[\mathrm{mov}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{MOV}_\chi\ x, y \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto R(y)], F\rangle}$$

$$[\mathrm{add}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{ADD}_\chi\ x, y, z \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{add}_{bin}(R(y), R(z), 0)], F\rangle}$$

$$[\mathrm{add}_{\mathrm{val}}] \qquad \frac{P(R(r_{pc})) = \mathsf{ADD}_\chi\ x, y, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{add}_{bin}(R(y), v, 0)], F\rangle}$$

$$[\mathrm{add}_{\mathrm{reg\text{-}set}}] \qquad \frac{P(R(r_{pc})) = \mathsf{ADDS}_\chi\ x, y, z \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{add}_{bin}(R(y), R(z), 0)], \mathit{flags}_{\mathsf{ADD}}(R(y), R(z))\rangle}$$

$$[\mathrm{add}_{\mathrm{val\text{-}set}}] \qquad \frac{P(R(r_{pc})) = \mathsf{ADDS}_\chi\ x, y, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{add}_{bin}(R(y), v, 0)], \mathit{flags}_{\mathsf{ADD}}(R(y), v)\rangle}$$

$$[\mathrm{sub}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{SUB}_\chi\ x, y, z \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{sub}_{bin}(R(y), R(z))], F\rangle}$$

$$[\mathrm{sub}_{\mathrm{val}}] \qquad \frac{P(R(r_{pc})) = \mathsf{SUB}_\chi\ x, y, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{sub}_{bin}(R(y), v)], F\rangle}$$

$$[\mathrm{sub}_{\mathrm{reg\text{-}set}}] \qquad \frac{P(R(r_{pc})) = \mathsf{SUBS}_\chi\ x, y, z \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{sub}_{bin}(R(y), R(z))], \mathit{flags}_{\mathsf{SUB}}(R(y), R(z))\rangle}$$

$$[\mathrm{sub}_{\mathrm{val\text{-}set}}] \qquad \frac{P(R(r_{pc})) = \mathsf{SUBS}_\chi\ x, y, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto \mathit{sub}_{bin}(R(y), v)], \mathit{flags}_{\mathsf{SUB}}(R(y), v)\rangle}$$

$$[\mathrm{cmp}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{CMP}_\chi\ x, y \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, \mathit{flags}_{\mathsf{SUB}}(R(x), R(y))\rangle}$$

$$[\mathrm{cmp}_{\mathrm{val}}] \qquad \frac{P(R(r_{pc})) = \mathsf{CMP}_\chi\ x, v \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, \mathit{flags}_{\mathsf{SUB}}(R(x), v)\rangle}$$

$$[\mathrm{ldr}_{\mathrm{addr}}] \qquad \frac{P(R(r_{pc})) = \mathsf{LDR}_\chi\ x, a \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto H(a)], F)\rangle}$$

$$[\mathrm{ldr}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{LDR}_\chi\ x, y \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto H(R(y))], F)\rangle}$$

$$[\mathrm{b}_{\mathrm{addr}}] \qquad \frac{P(R(r_{pc})) = \mathsf{B}_\chi\ a \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R[r_{pc} \mapsto a], F)\rangle}$$

$$[\mathrm{b}_{\mathrm{reg}}] \qquad \frac{P(R(r_{pc})) = \mathsf{B}_\chi\ x \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R[r_{pc} \mapsto R(x)], F)\rangle}$$

$$[\text{str}_{\text{addr}}] \quad \frac{P(R(r_{pc})) = \texttt{STR}_\chi\ x, a \quad cond(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H[a \mapsto R(x)], R_{r_{pc}+1}, F)\rangle}$$

$$[\text{str}_{\text{reg}}] \quad \frac{P(R(r_{pc})) = \texttt{STR}_\chi\ x, y \quad cond(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H[R(y) \mapsto R(x)], R_{r_{pc}+1}, F)\rangle}$$

$$[\text{nop}] \quad \frac{P(R(r_{pc})) = \texttt{NOP}_\chi \quad cond(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, F)\rangle}$$

$$[\text{skip}] \quad \frac{P(R(r_{pc})) = instr_\chi \quad \neg cond(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, F)\rangle}$$

where $C \Longrightarrow C'$ is the reduction relation between configurations, $C, C' \in \text{Conf}$.

We further define $\Longrightarrow^n$ as a sequence of n reductions.

A.2 Fault Models

We formulate specific semantic rules for the different types of faults that can occur. In these rules we annotate the reduction relation with the type of fault that has occurred:

$$C \Longrightarrow_\phi C'$$

where $C, C' \in \text{Conf}$, $\phi \in \mathcal{F}$ and $\mathcal{F}$ is the fault model, i.e. the set of faults that can occur. Similar to $\Longrightarrow^n$, $\Longrightarrow^n_\phi$ describes a sequence of n reductions where one fault, ϕ, has occurred at some point.

We need a formal way of describing what happens to a bit string when an Single Event Upset (SEU) occurs. We define what it means for two bit strings, $v_1, v_2 \in \mathbb{B}_n$ to differ by exactly one bit, i.e. have a Hamming distance 1:

$$v_1 \equiv_1 v_2 \quad \text{iff} \quad \exists i\ \forall j\ (i, j \in [0..n-1] \wedge v_1(i) \neq v_2(j) \iff i = j)$$

All fault models are prefixed with f- to differentiate them from the general semantic rules. We create two rules for SEUs in registers, one for faults in any of the general registers and one for faults in a control registers:

$$[f\text{-reg}_{\text{gen}}] \quad \frac{x \in \text{GeneralRegister} \qquad v = R(x) \qquad v' \equiv_1 v}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-reg}_{\text{gen}}} \langle P, H, R[x \mapsto v'], F\rangle}$$

$$[f\text{-reg}_{\text{ctrl}}] \quad \frac{x \in \text{ControlRegister} \qquad v = R(x) \qquad v' \equiv_1 v}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-reg}_{\text{ctrl}}} \langle P, H, R[x \mapsto v'], F\rangle}$$

We define a rule for SEUs changing the values stored in a control flag:

$$[f\text{-flag}] \quad \frac{f \in \text{Flag} \qquad b = F(f) \qquad b' = \bar{b}}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-flag}} \langle P, H, R, F[f \mapsto b']\rangle}$$

We also define a rule for how a SEU can change a single instruction just before it is executed. For this, we require the ability to talk about the binary encoding of each instruction (their opcodes), such that we can talk about which instructions have a Hamming distance of one. We simply define the function $opcode\colon Instr \to \mathbb{B}_{32}$ to return the binary encoding of an instruction. The rule can then be defined as:

$$[f\text{-instr}] \quad \frac{\begin{array}{c} instr = P(R(r_{pc})) \qquad opcode(instr') \equiv_1 opcode(instr) \\ P' = P[R(r_{pc}) \mapsto instr'] \\ \langle P', H, R, F\rangle \Longrightarrow \langle P', H', R', F'\rangle \end{array}}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-instr}} \langle P, H', R', F'\rangle}$$

We define how SEUs in memory would behave. Memory encompasses both the program itself and all the data stored in the heap, but we faults in them as separate faults:

$$[f\text{-mem}_{\text{prog}}] \quad \frac{a \in \mathbf{Addr} \qquad instr = P(a) \qquad opcode(instr') \equiv_1 opcode(instr)}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-mem}_{\text{prog}}} \langle P[a \mapsto instr'], H, R, F\rangle}$$

$$[f\text{-mem}_{\text{data}}] \quad \frac{a \in \mathbf{Addr} \qquad v = H(a) \qquad v' \equiv_1 v}{\langle P, H, R, F\rangle \Longrightarrow_{f\text{-mem}_{\text{data}}} \langle P, H[a \mapsto v'], R, F\rangle}$$

Finally, we define the fault that any one instruction is skipped.

$$[f\text{-skip}] \quad \langle P, H, R, F\rangle \Longrightarrow_{f\text{-skip}} \langle P, H, R_{r_{pc}+1}, F\rangle$$

B Interval Analysis

The analysis lattice L and supporting definitions:

$$\begin{array}{l}
L = (\mathcal{P}(L_f \times Vars), \subseteq) \\
L_f = L_{I_1} \times L_{I_1} \times L_{I_1} \times L_{I_1} \\
Vars = GeneralRegister \to L_{i_{MAX}} \\
L_{I_n} = lift((\{[a..b] \mid a, b \in \mathbb{N}_n, a \leq b\}, \sqsubseteq_{I_n})) \\
\sqsubseteq_{I_n} = \{([l_1..r_1], [l_2..r_2]) \mid [l_1..r_1], [l_2..r_2] \in I_n, l_1 \geq l_2, r_1 \leq r_2\}
\end{array}$$

$$\begin{array}{l}
\mathbb{N}_x = \{n \mid n \in \mathbb{N}, n \leq x\} \\
MAX = 2^{32} - 1
\end{array}$$

The *in* and *out* functions:

$$in(n) = \bigsqcup\{out(p) \mid p \in pred(n)\}$$

$$out(n) = \begin{cases}
t_{\mathsf{MOV}v}(r, v, in(n)) & \text{if } n = \mathsf{MOV}_\chi\ r, v \\
t_{\mathsf{MOV}r}(r_{dst}, r_{src}, in(n)) & \text{if } n = \mathsf{MOV}_\chi\ r_{dst}, r_{src} \\
t_{\mathsf{LDR}}(r, in(n)) & \text{if } n = \mathsf{LDR}_\chi\ r, \_ \\
t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, in(n)) & \text{if } n = \mathsf{ADD}_\chi\ r_{dst}, r_{src}, v \\
t_{\mathsf{ADD}r}(r_{dst}, r_{src1}, r_{src2}, in(n)) & \text{if } n = \mathsf{ADD}_\chi\ r_{dst}, r_{src1}, r_{src2} \\
t_{\mathsf{ADDS}v}(r_{dst}, r_{src}, v, in(n)) & \text{if } n = \mathsf{ADDS}_\chi\ r_{dst}, r_{src}, v \\
t_{\mathsf{ADDS}r}(r_{dst}, r_{src1}, r_{src2}, in(n)) & \text{if } n = \mathsf{ADDS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\
t_{\mathsf{SUB}v}(r_{dst}, r_{src}, v, in(n)) & \text{if } n = \mathsf{SUB}_\chi\ r_{dst}, r_{src}, v \\
t_{\mathsf{SUB}r}(r_{dst}, r_{src1}, r_{src2}, in(n)) & \text{if } n = \mathsf{SUB}_\chi\ r_{dst}, r_{src1}, r_{src2} \\
t_{\mathsf{SUBS}v}(r_{dst}, r_{src}, v, in(n)) & \text{if } n = \mathsf{SUBS}_\chi\ r_{dst}, r_{src}, v \\
t_{\mathsf{SUBS}r}(r_{dst}, r_{src1}, r_{src2}, in(n)) & \text{if } n = \mathsf{SUBS}_\chi\ r_{dst}, r_{src1}, r_{src2} \\
t_{\mathsf{CMP}v-}(r, v, in(n)) & \text{if } n = \mathsf{CMP}_\chi\ r, v \\
t_{\mathsf{CMP}r-}(r_x, r_y, in(n)) & \text{if } n = \mathsf{CMP}_\chi\ r_x, r_y \\
t_{\mathsf{ASSERT}}(\chi, in(n)) & \text{if } n = \mathsf{ASSERT}\chi \\
t_{\mathsf{REFUTE}}(\chi, in(n)) & \text{if } n = \mathsf{REFUTE}\chi \\
in(n) & \text{otherwise}
\end{cases}$$

Transfer functions:

$t_{\mathsf{MOV}v} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{MOV}v}(r, v, l) = \{(f, i[r \mapsto [v..v]]) \mid (f, i) \in l\}$

$t_{\mathsf{MOV}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{MOV}r}(r_{dst}, r_{src}, l) = \{(f, i[r_{dst} \mapsto i(r_{src})]) \mid (f, i) \in l\}$

$t_{\mathsf{LDR}} : \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{LDR}}(r, l) = \{(f, i[r \mapsto [0..MAX_{bw}]]) \mid (f, i) \in l\}$

$t_{\mathsf{ADD}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, l) = \{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{src}), [v..v])]) \mid (f, i) \in l\}$

$t_{\mathsf{ADD}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{ADD}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l) = \{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{\mathit{src1}}), i(r_{\mathit{src2}}))]) \mid (f, i) \in l\}$

$t_{\mathsf{ADDS}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{ADDS}v}(r_{dst}, r_{src}, v, l) = t_{\mathsf{CMP}v+}(r_{src}, v, t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, l))$

$t_{\mathsf{ADDS}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{ADDS}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l) = t_{\mathsf{CMP}r+}(r_{\mathit{src1}}, r_{\mathit{src2}}, t_{\mathsf{ADD}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l))$

$t_{\mathsf{SUB}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{SUB}v}(r_{dst}, r_{src}, v, l) = \{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{src}), [v..v])]) \mid (f, i) \in l\}$

$t_{\mathsf{SUB}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{SUB}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l) = \{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{\mathit{src1}}), i(r_{\mathit{src2}}))]) \mid (f, i) \in l\}$

$t_{\mathsf{SUBS}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{SUBS}v}(r_{dst}, r_{src}, v, l) = t_{\mathsf{CMP}v-}(r_{src}, v, t_{\mathsf{SUB}v}(r_{dst}, r_{src}, v, l))$

$t_{\mathsf{SUBS}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{SUBS}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l) = t_{\mathsf{CMP}r-}(r_{\mathit{src1}}, r_{\mathit{src2}}, t_{\mathsf{SUB}r}(r_{dst}, r_{\mathit{src1}}, r_{\mathit{src2}}, l))$

$t_{\mathsf{ASSERT}} : \mathit{Condition} \times L \times L$
$t_{\mathsf{ASSERT}}(\chi, l) = \{(f, i) \mid (f, i) \in l, \mathit{matchesCondition}(f, \chi)\}$

$t_{\mathsf{REFUTE}} : \mathit{Condition} \times L \to L$
$t_{\mathsf{REFUTE}}(\chi, l) = \{(f, i) \mid (f, i) \in l, \mathit{matchesCondition}(f, \neg\chi)\}$

$t_{\mathsf{CMP}v-} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{CMP}v-}(r, v, l) = \{(f', i[r \mapsto a]) \mid (\_, i) \in l, (f', a, \_) \in \mathit{split}_{sub}(i(r), [v..v])\}$

$t_{\mathsf{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{CMP}r-}(r_x, r_y, l) = \{(f', i[r_x \mapsto a][r_y \mapsto b]) \mid (\_, i) \in l, (f', a, b) \in \mathit{split}_{sub}(i(r_x), i(r_y))\}$

$t_{\mathsf{CMP}v+} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L$
$t_{\mathsf{CMP}v+}(r, v, l) = \{(f', i[r \mapsto a]) \mid (\_, i) \in l, (f', a, \_) \in \mathit{split}_{add}(i(r), [v..v])\}$

$t_{\mathsf{CMP}r+} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times L \to L$
$t_{\mathsf{CMP}r+}(r_x, r_y, l) = \{(f', i[r_x \mapsto a][r_y \mapsto b]) \mid (\_, i) \in l, (f', a, b) \in \mathit{split}_{add}(i(r_x), i(r_y))\}$

Supporting functions:

$$
\begin{aligned}
&\mathit{add}_I : L_{I_{MAX}} \times L_{I_{MAX}} \rightarrow L_{I_{MAX}}\\
&\mathit{add}_I(\bot, \_) = \bot\\
&\mathit{add}_I(\_, \bot) = \bot\\
&\mathit{add}_I([l_1..r_1],[l_2..r_2]) = \begin{cases} [0..MAX] & \text{if } (l_1 + l_2) \leq MAX \wedge (r_1 + r_2) > MAX \\ [\mathit{add}_{bin}(l_1, l_2, 0)..\mathit{add}_{bin}(r_1, r_2, 0)] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sub}_I : L_{I_{MAX}} \times L_{I_{MAX}} \rightarrow L_{I_{MAX}}\\
&\mathit{sub}_I(\bot, \_) = \bot\\
&\mathit{sub}_I(\_, \bot) = \bot\\
&\mathit{sub}_I([l_1..r_1],[l_2..r_2]) = \begin{cases} [0..MAX] & \text{if } (l_1 - r_2) < 0 \wedge (r_1 - l_2) \geq 0 \\ [\mathit{sub}_{bin}(l_1, r_2)..\mathit{sub}_{bin}(r_1, l_2)] & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{matchesCondition} : L_f \times \mathit{ConditionCode} \rightarrow \{\mathit{true}, \mathit{false}\}\\
&\mathit{matchesCondition}((n, z, c, v), \chi) =\\
&\qquad\begin{cases}
[1..1] \sqsubseteq z & \text{if } \chi = \texttt{EQ}\\
[0..0] \sqsubseteq z & \text{if } \chi = \texttt{NE}\\
[1..1] \sqsubseteq c & \text{if } \chi = \texttt{CS}\\
[0..0] \sqsubseteq c & \text{if } \chi = \texttt{CC}\\
[1..1] \sqsubseteq n & \text{if } \chi = \texttt{MI}\\
[0..0] \sqsubseteq n & \text{if } \chi = \texttt{PL}\\
[1..1] \sqsubseteq v & \text{if } \chi = \texttt{VS}\\
[0..0] \sqsubseteq v & \text{if } \chi = \texttt{VC}\\
[0..0] \sqsubseteq z \wedge [1..1] \sqsubseteq c & \text{if } \chi = \texttt{HI}\\
[0..0] \sqsubseteq c \vee [1..1] \sqsubseteq z & \text{if } \chi = \texttt{LS}\\
([1..1] \sqsubseteq n \wedge [1..1] \sqsubseteq v) \vee ([0..0] \sqsubseteq n \wedge [0..0] \sqsubseteq v) & \text{if } \chi = \texttt{GE}\\
([1..1] \sqsubseteq n \wedge [0..0] \sqsubseteq v) \vee ([0..0] \sqsubseteq n \wedge [1..1] \sqsubseteq v) & \text{if } \chi = \texttt{LT}\\
([1..1] \sqsubseteq n \wedge [0..0] \sqsubseteq z \wedge [1..1] \sqsubseteq v) \vee ([0..0] \sqsubseteq n \wedge [0..0] \sqsubseteq z \wedge [0..0] \sqsubseteq v) & \text{if } \chi = \texttt{GT}\\
([1..1] \sqsubseteq z) \vee ([1..1] \sqsubseteq n \wedge [0..0] \sqsubseteq v) \vee ([0..0] \sqsubseteq n \wedge [1..1] \sqsubseteq v) & \text{if } \chi = \texttt{LE}\\
\mathit{true} & \text{if } \chi = \texttt{AL}\\
\mathit{false} & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{split}_{sub} : L_{I_{MAX}} \times L_{I_{MAX}} \rightarrow \mathcal{P}(L_f \times L_{I_{MAX}} \times L_{I_{MAX}})\\
&\mathit{split}_{sub}(i_1, i_2) = \mathit{combine}(\{((f(f_N), f(f_Z), f(f_C), f(f_v)), [a..a], [b..b]) \mid a \in i_1, b \in i_2, f = \mathit{flags}_{\texttt{ADD}}(a, b)\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{split}_{sub} : L_{I_{MAX}} \times L_{I_{MAX}} \rightarrow \mathcal{P}(L_f \times L_{I_{MAX}} \times L_{I_{MAX}})\\
&\mathit{split}_{sub}(i_1, i_2) = \mathit{combine}(\{((f(f_N), f(f_Z), f(f_C), f(f_v)), [a..a], [b..b]) \mid a \in i_1, b \in i_2, f = \mathit{flags}_{\texttt{SUB}}(a, b)\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{combine} : P(L_f \times L_{I_{MAX}} \times L_{I_{MAX}}) \rightarrow P(L_f \times L_{I_{MAX}} \times L_{I_{bw}})\\
&\mathit{combine}(X) = \{(f, [\mathit{min}(L_1)..\mathit{max}(R_1)], [\mathit{min}(L_2)..\mathit{max}(R_2)]) \mid\\
&\qquad L_1 = \{\, l \mid (f, [l..\_], \_) \in X \,\},\\
&\qquad R_1 = \{\, r \mid (f, [\_..r], \_) \in X \,\},\\
&\qquad L_2 = \{\, l \mid (f, \_, [l..\_]) \in X \,\},\\
&\qquad R_2 = \{\, r \mid (f, \_, [\_..r]) \in X \,\}\\
&\}
\end{aligned}
$$

C Bitflip Analysis

The analysis lattice L and supporting definitions.

$$\begin{aligned}
&L = K \to V\\
&K = (\mathit{Label} \times \mathit{GeneralRegister}) \cup \{\omega\}\\
&V = \mathcal{P}(L_f \times \mathit{Vars} \times B)\\
&\mathit{Vars} = \mathit{GeneralRegister} \to L_{i_{MAX}}\\
&B = \mathcal{P}(\mathbb{N}_{bw-1})^2
\end{aligned}$$

The *in* and *out* functions:

$$\mathit{in}(n) = \bigsqcup\{\mathit{out}(p) \mid p \in \mathit{pred}(n)\}$$

$$\mathit{out}(n) = \begin{cases}
t_{\mathsf{MOV}v}(r, v, \mathit{in}(n)) & \text{if } n = \mathsf{MOV}_\chi\ r, v\\
t_{\mathsf{MOV}r}(r_{dst}, r_{src}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{MOV}_\chi\ r_{dst}, r_{src}\\
t_{\mathsf{LDR}v}(r, \mathit{in}(n)) & \text{if } n = \mathsf{LDR}_\chi\ r, v\\
t_{\mathsf{LDR}r}(r_{dst}, r_{addr}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{LDR}_\chi\ r_{dst}, r_{addr}\\
t_{\mathsf{STR}}(\{r\}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{STR}_\chi\ r, v\\
t_{\mathsf{STR}}(\{r_x, r_y\}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{STR}_\chi\ r_x, r_y\\
t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{ADD}_\chi\ r_{dst}, r_{src}, v\\
t_{\mathsf{ADD}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{ADD}_\chi\ r_{dst}, r_{src1}, r_{src2}\\
t_{\mathsf{ADDS}v}(r_{dst}, r_{src}, v, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{ADDS}_\chi\ r_{dst}, r_{src}, v\\
t_{\mathsf{ADDS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{ADDS}_\chi\ r_{dst}, r_{src1}, r_{src2}\\
t_{\mathsf{SUB}v}(r_{dst}, r_{src}, v, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{SUB}_\chi\ r_{dst}, r_{src}, v\\
t_{\mathsf{SUB}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{SUB}_\chi\ r_{dst}, r_{src1}, r_{src2}\\
t_{\mathsf{SUBS}v}(r_{dst}, r_{src}, v, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{SUBS}_\chi\ r_{dst}, r_{src}, v\\
t_{\mathsf{SUBS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{SUBS}_\chi\ r_{dst}, r_{src1}, r_{src2}\\
t_{\mathsf{CMP}v-}(r, v, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{CMP}_\chi\ r, v\\
t_{\mathsf{CMP}r-}(r_x, r_y, \mathit{label}(n), \mathit{in}(n)) & \text{if } n = \mathsf{CMP}_\chi\ r_x, r_y\\
t_{\mathsf{ASSERT}}(\chi, \mathit{in}(n)) & \text{if } n = \mathsf{ASSERT}\ \chi\\
t_{\mathsf{REFUTE}}(\chi, \mathit{in}(n)) & \text{if } n = \mathsf{REFUTE}\ \chi\\
\mathit{in}(n) & \text{otherwise}
\end{cases}$$

The individual transfer functions

$$\begin{array}{l} t_{id} : L \to L \\ t_{id}(l) = l \end{array}$$

$$\begin{array}{l} t_{\mathsf{MOV}v} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times L \to L \\ t_{\mathsf{MOV}v}(r, n, l) \mathit{applyAll}(\lambda(X).\{(f, i[r \mapsto [n..n]], b) \mid (f, i, b) \in X\})(l) \end{array}$$

$$\begin{array}{l} t_{\mathsf{MOV}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\ t_{\mathsf{MOV}r}(r_{dst}, r_{src}, \mathit{lbl}, l) = \mathit{applyAll}(\lambda(X).\{(f, i[r_{dst} \mapsto i(r_{src})], b) \mid (f, i, b) \in X\})(\mathit{flip}(t_{id}, \mathit{lbl}, \{r_{src}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{LDR}v} : \mathit{GeneralRegister} \times L \to L \\ t_{\mathsf{LDR}v}(r, l) = \mathit{applyAll}(\lambda(X).\{(f, i[r \mapsto [0..MAX]], b) \mid (f, i, b) \in X\})(l) \end{array}$$

$$\begin{array}{l} t_{\mathsf{LDR}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\ t_{\mathsf{LDR}r}(r_{dst}, r_{addr}, \mathit{lbl}, l) = \mathit{applyAll}(\lambda(X).\{(f, i[r \mapsto [0..MAX]], b) \mid (f, i, b) \in X\})(\mathit{flip}(t_{id}, \mathit{lbl}, \{r_{addr}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{STR}} : \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \to L \\ t_{\mathsf{STR}}(\mathit{lbl}, R, l) = \mathit{flip}(\mathit{id}, \mathit{lbl}, R, l) \end{array}$$

$$\begin{array}{l} t_{\mathsf{ADD}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L \\ t_{\mathsf{ADD}v}(r_{dst}, r_{src}, v, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{src}), [v..v])], b) \mid (f, i, b) \in X\})(\\ \qquad \mathit{flip}(t_{id}, \mathit{lbl}, \{r_{src}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{ADD}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\ t_{\mathsf{ADD}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{src1}), i(r_{src2}))], b) \mid (f, i, b) \in X\})(\\ \qquad \mathit{flip}(t_{id}, \mathit{lbl}, \{r_{src1}, r_{src2}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{ADDS}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L \\ t_{\mathsf{ADDS}v}(r_{dst}, r_{src}, v, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{src}), [v..v])], b) \mid (f, i, b) \in X\})(\\ \qquad \mathit{flip}(t'_{\mathsf{CMPv+}}(r_{src}, v), \mathit{lbl}, \{r_{src}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{ADDS}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\ t_{\mathsf{ADDS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{add}_I(i(r_{src1}), i(r_{src2}))], b) \mid (f, i, b) \in X\})(\\ \qquad (\mathit{flip}(t'_{\mathsf{CMPr+}}(r_{src1}, r_{src2}), \mathit{lbl}, \{r_{src1}, r_{src2}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{SUB}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L \\ t_{\mathsf{SUB}v}(r_{dst}, r_{src}, v, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{src}), [v..v])], b) \mid (f, i, b) \in X\})(\\ \qquad \mathit{flip}(t_{id}, \mathit{lbl}, \{r_{src1}\}, l)) \end{array}$$

$$\begin{array}{l} t_{\mathsf{SUB}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L \\ t_{\mathsf{SUB}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{lbl}, l) = \mathit{applyAll}(\\ \qquad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{src1}), i(r_{src2}))], b) \mid (f, i, b) \in X\})(\\ \qquad \mathit{flip}(t_{id}, \mathit{lbl}, \{r_{src1}, r_{src2}\}, l)) \end{array}$$

$t_{\mathsf{SUBS}v} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L$
$t_{\mathsf{SUBS}v}(r_{dst}, r_{src}, v, l) = \mathit{applyAll}($
$\lambda(X).\{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{src}), [v..v])], b) \mid (f, i, b) \in X\})($
$\quad \mathit{flip}(t'_{\mathsf{CMP}v-}(r_{src}, v), \mathit{lbl}, \{r_{src}\}, l))$

$t_{\mathsf{SUBS}r} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L$
$t_{\mathsf{SUBS}r}(r_{dst}, r_{src1}, r_{src2}, \mathit{lbl}, l) = \mathit{applyAll}($
$\quad \lambda(X).\{(f, i[r_{dst} \mapsto \mathit{sub}_I(i(r_{src1}), i(r_{src2}))], b) \mid (f, i, b) \in X\})($
$\quad \mathit{flip}(t'_{\mathsf{CMP}r-}(r_{src1}, r_{src2}), \mathit{lbl}, \{r_{src1}, r_{src2}\}, l))$

$t_{\mathsf{ASSERT}} : \mathit{Condition} \times L \to L$
$t_{\mathsf{ASSERT}}(\chi, l) = \mathit{applyAll}(\lambda(X).\{(f, i, b) \mid (f, i, b) \in X, \mathit{matchesCondition}(f, \chi)\})(l)$

$t_{\mathsf{REFUTE}} : \mathit{Condition} \times L \to L$
$t_{\mathsf{REFUTE}}(\chi, l) = \mathit{applyAll}(\lambda(X).\{(f, i, b) \mid (f, i, b) \in X, \mathit{matchesCondition}(f, \neg\chi)\})(l)$

$t_{\mathsf{CMP}v-} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L$
$t_{\mathsf{CMP}v-}(r, v, \mathit{lbl}, l) = \mathit{flip}(t'_{\mathsf{CMP}v-}(r, v), \mathit{lbl}, \{r\}, l)$

$t'_{\mathsf{CMP}v-} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \to L \to L$
$t'_{\mathsf{CMP}v-}(r, v) = \mathit{applyAll}(\lambda(X).\{(f', i[r \mapsto a], b) \mid (\_, i, b) \in X, (f', a, \_) \in \mathit{split}_{sub}(i(r), [v..v])\})$

$t_{\mathsf{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L$
$t_{\mathsf{CMP}r-}(r_x, r_y, \mathit{lbl}, l) = \mathit{flip}(t'_{\mathsf{CMP}r-}(r_x, r_y), \mathit{lbl}, \{r_x, r_y\}, l)$

$t'_{\mathsf{CMP}r-} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \to L \to L$
$t'_{\mathsf{CMP}r-}(r_x, r_y) = \mathit{applyAll}(\lambda(X).\{(f', i[r_x \mapsto s_x][r_y \mapsto s_y], b) \mid (\_, i, b) \in X, (f', s_x, s_y) \in \mathit{split}_{\mathsf{sub}}(i(r_x), i(r_y))\})$

$t_{\mathsf{CMP}v+} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \times \mathit{Label} \times L \to L$
$t_{\mathsf{CMP}v+}(r, v, \mathit{lbl}, l) = \mathit{flip}(t'_{\mathsf{CMP}v+}(r, v), \mathit{lbl}, \{r\}, l)$

$t'_{\mathsf{CMP}v+} : \mathit{GeneralRegister} \times \mathbb{N}_{MAX} \to L \to L$
$t'_{\mathsf{CMP}v+}(r, v) = \mathit{applyAll}(\lambda(X).\{(f', i[r \mapsto s], b) \mid (\_, i, b) \in X, (f', s, \_) \in \mathit{split}_{add}(i(r), [v..v])\})$

$t_{\mathsf{CMP}r+} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \times \mathit{Label} \times L \to L$
$t_{\mathsf{CMP}r+}(r_x, r_y, \mathit{lbl}, l) = \mathit{flip}(t'_{\mathsf{CMP}r+}(r_x, r_y), \mathit{lbl}, \{r_x, r_y\}, l)$

$t'_{\mathsf{CMP}r+} : \mathit{GeneralRegister} \times \mathit{GeneralRegister} \to L \to L$
$t'_{\mathsf{CMP}r+}(r_x, r_y) = \mathit{applyAll}(\lambda(X).\{(f', i[r_x \mapsto s_x][r_y \mapsto s_y], b) \mid (\_, i, b) \in X, (f', s_x, s_y) \in \mathit{split}_{add}(i(r_x), i(r_y))\})$

Supporting functions

$$
\begin{aligned}
&\mathit{applyAll} : (V \to V) \to L \to L \\
&\mathit{applyAll}(f)(l) = \lambda(k).f(l(k)) \\
&\mathit{flip} : (L \to L) \times \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \to L \\
&\mathit{flip}(t, \mathit{lbl}, R, l) = \mathit{flip}'(\mathit{lbl}, R, l, t(\mathit{expand}(\mathit{lbl}, R, l)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flip}' : \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \times L \to L \\
&\mathit{flip}'(\mathit{lbl}, R, l_{pre}, l_{post}) = \lambda(k).\begin{cases} (f', i', \mathit{possibleFlips}(i(r), i'(r))) & \text{if } (\mathit{lbl}, r) \in k \wedge r \in R \\ l_{post}(k) & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{where } (f, i, \_) \in l_{pre}(k), (f', i', \_) \in l_{post}(k)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{expand} : \mathit{Label} \times \mathcal{P}(\mathit{GeneralRegister}) \times L \to L \\
&\mathit{expand}(\mathit{lbl}, R, l) = \lambda(\mathit{lbl}', r).\begin{cases} \mathit{expand}'(r, l(\omega)) & \text{if } r \in R \wedge \mathit{lbl} = \mathit{lbl}' \\ l(\mathit{lbl}', r) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{expand}' : \mathit{GeneralRegister} \times V \to V \\
&\mathit{expand}'(r, (f, i, b)) = (f, i[r \mapsto \mathit{widenInterval}(i(r))], b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{widenInterval} : L_{i_{MAX}} \to L_{i_{MAX}} \\
&\mathit{widenInterval}(\bot) = \bot \\
&\mathit{widenInterval}([l..r]) = \dots
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{possibleFlips} : L_{i_{MAX}} \times L_{i_{MAX}} \to B \\
&\mathit{possibleFlips}(\bot, \_) = \emptyset \\
&\mathit{possibleFlips}(\_, \bot) = \emptyset \\
&\mathit{possibleFlips}([l_{pre}..r_{pre}], [l_{post}..r_{post}]) = \dots
\end{aligned}
$$