Summary

This report presents a formalization of a small ARM-based assembly language with explicit structure for modelling faults on specific bit positions and registers. More specifically, we model single event upsets that can be caused by various natural phenomena, or by malicious adversaries exploiting bugs, such as the rowhammer bug. We leverage data/control flow analysis to obtain a program slice towards a user-specified critical program point. The critical program point is annotated with a special security assertion which is a logic formula used to verify the safety/security properties of the program.

We present a formal static program analysis based on symbolic execution, which obtains the path conditions on traces in the program slice to the critical program point. The collected path conditions are in the form of logic formulas, and we utilize the Z3 SMT solver to decide whether path conditions conjoined with the security assertion are satisfiable. The satisfiability implies that a fault can break the security/safety properties of the program, and therefore causes a vulnerability in the system. The formalisation of the analysis is an extension of the structural operational semantics of the targeted assembly language.

Furthermore, we observe that the effects of faults occurring at different registers and bit positions at different time instances during program execution may yield the same effect on program behaviour. Since it is more practical to reason about the possible effects of faults on program behaviour, than to reason about individual faults, we propose to categorize the set of all possible faults into *fault equivalence classes* conditioned on their effect on program behaviour. We present a formal definition of fault equivalence in terms of whether the conditional flags may differ at program points during execution with each fault. Additionally, we propose an algorithm for deciding whether two faults are equivalent. The algorithm leverages the path conditions already collected by the previous symbolic execution step, and attempts, for each program point, to prove that the flags cannot differ in two program executions with one fault occurring in each execution.

We show how programs with fault instructions can be modelled as timed automata. This enables us to analyse and quantify programs with *statistical model checking*. We leverage the statistical model checking capabilities of the model checking tool UPPAAL.

To showcase the analysis, we conduct experiments on a simple controller program. More specifically, we analyse the program to obtain the quantification of the risk for each vulnerable fault found by the vulnerability analysis. We then apply a simple instruction duplication scheme and perform experiments on this hardened version of the program to compare.

Finally, we discuss how the analysis may be extended to include loops, which were assumed not to be present in the initial formalisation.

Quantitative Analysis of Single Event Upsets in ARM

Thomas Rafn Andersen, Morten Korsholm Terndrup
{tran13, mternd13}@student.aau.dk

Supervisor: Rene Rydhof Hansen

Department of Computer Science
Aalborg University
Selma Lagerløfs Vej 300, 9220 Aalborg Ø

June 14, 2018

Abstract

We propose a static program analysis for quantifying the vulnerability of programs in the presence of single event upsets. The analysis is formalised and implemented for an ARM based assembly language, in which we provide syntax and semantics for modelling faults. The analysis itself targets a user-specified critical program point and utilizes data and control flow analysis to obtain a relevant program slice. The analysis leverages symbolic execution to collect the path conditions on the slice towards the critical code, and the vulnerable faults are obtained by verifying path conditions against a special security assertion. We show how program semantics can be modelled as timed automata models, allowing us to analyse and quantify the risk of faults with statistical model checking. Furthermore, due to the many possible ways single event upsets can affect a program, we propose to analyse faults in terms of *fault equivalence classes* conditioned on their effect on the behaviour of a program. We propose a definition of fault equivalence and provide an algorithm for deciding equivalence. Lastly, we show the usefulness of the analysis by experiments with an implementation of a simple prototype tool.

1 Introduction

In recent years, we have seen a large increase in the demand for smaller, faster, and more power efficient hardware. Consequently, hardware with great performance for the use in hand-held and IoT devices are becoming smaller with a much higher transistor density. However, the lower voltages of this hardware makes it less reliable and more susceptible errors [10, 7]. These hardware errors are known as *soft errors* or *transient faults* in the system. Such faults will not cause permanent damage, but they can have a temporary effect on the execution of software that is currently running on the system and therefore the overall behaviour of the system. In safety and security critical systems, the consequences of such faults may be catastrophic.

One type of soft error is the *Single Event Upset* (SEU), which is a fault happening in the cache or memory of a system that causes a single bit in a single register, during one execution of a program, to *flip* (i.e. a zero becomes a one, or vice versa). Such a fault can happen when electronic components are exposed to environmental hazards like cosmic rays or high energy protons. One of the early reports of such errors came after packaging materials for CPU chips were contaminated by a nearby uranium mine. When particles from the contamination struck the die they could cause SEUs[12]. While a single bit changing may not sound dangerous, it can cause various errors to happen. For instance causing a spacecraft to enter "safe mode"[1]. These types of faults therefore have the potential of causing great economic loss or, in the case of a spacecraft, ruin important, and expensive, multi-year missions. In extreme cases such faults could even be dangerous, *you would want a nuclear power plant to warn you about a potential meltdown, wouldn't you?*

Before we get into the detail of our report, we illustrate how SEU faults can influence program execution with a motivating example.

Example 1
Lets consider the simple alarm controller program in Listing 1.

```c
short temp = read(0xFEED);
short min = 0, max = 10000, danger = 2000;
if (temp < min || temp > max)
    error();
else if (temp >= danger)
    alarm();
else
    safe();
```

Listing 1: Alarm controller program in C.

The program reads a value from a sensor and takes action based on the value returned by the sensor. This value could be the temperature reading from some unreliable sensor in a larger system. Three cases are considered in the program: (1) the sensor returned an error value, (2) the sensor returned a dangerous value which should trigger the alarm, or (3) the sensor returned a safe value and nothing should happen.

The program contains four variables which are susceptible to SEU faults, some of which are able to change the program's execution flow. Here are some examples:

- The safe value 1500 is read, and it is flipped to 3548 (flip bit 11).

  Result: The alarm is sounded when it should not have.

- The dangerous value 2500 is read, and the danger variable is flipped to 4048 (flip bit 11).

  Result: The input is assumed safe when it is not and no alarm is triggered.

These are just two possible SEU faults that can change the program's execution, but there are many other faults like these.

[1]`https://www.nasa.gov/mission_pages/cassini/whycassini/cassini20101109.html`

While hardware based solutions exist for the problem of SEUs (such as memory with ECC), we propose a software based static program analysis for determining whether an SEU can break the safety/security of a program. A software based solution has the advantage that it is more flexible and can be used without replacing existing hardware. The analysis is based on our previous work[2], and it combines data and control flow analysis with directed symbolic execution to analyse the effects of different SEU faults on the execution of a program. To facilitate this, we utilise special *security assertions* that are used in symbolic execution to detect contradictions between register values present at given program points and at the critical program point.

Furthermore, we present a novel idea of grouping faults by their effect on program behaviour into a set of *fault equivalence classes*. Grouping faults in this manner provides a meaningful abstraction over the complexity of reasoning about individual faults, while still providing much better insights into the characteristics of the fault than conventional black-box fault injection methods. We formally define fault equivalence and provide an algorithm for deciding the equivalence of faults.

Finally, we apply statistical model checking through UPPAAL SMC and run simulations on the faults we found to be vulnerable during the analysis. This allows us to estimate the risk for various events that can happen as a result of a fault.

In summary, we consider the following to be our main contributions:

- The formalisation of a static program analysis that combines data and control flow analysis with symbolic execution to determine program points vulnerable to SEU faults.
- A novel idea and formalisation of fault equivalence as well as a procedure for deciding fault equivalence.
- How programs and faults can be modelled, analysed, and quantified by encoding them as timed automata and leveraging statistical model checking.

Before we give an overview of the report, we have a quick note about the use of the words security and safety. The analysis can be used for both safety and security critical systems, but for brevity we use safety and security interchangeably throughout the report. This means that any of the words covers the meaning of both in most circumstances. This is especially true when we talk about security assertions.

The report is structured in the following way. In Section 2 we present related work. In Section 3 we show the changes done to the language from our previous work as well as the fault model we are using. In Section 4 we present our symbolic execution semantics, which again are based on our previous work. In Section 5 we discuss SEU fault equivalence. In Section 6 we give an overview of our analysis. In Section 7 we describe the program representations we are using. In Section 8 we show how we choose the SEU faults that the analysis works on. In Section 9 we explain our quantitative analysis. In Section 10 we perform experiments with our analysis. In Section 11 we discuss how our analysis can be extended to handle programs with loops. In Section 12 we conclude and discuss possible future work.

2 Related Work

Transient faults do not only occur by natural phenomena. Kim et al. [10] discovered that when the rows of modern DRAM modules are accessed in quick succession data may leak into adjacent rows. This phenomenon is known as the *rowhammer* bug. They showed that rowhammer can easily be induced in x86 architecture by non-privileged cache flush instructions.

Since rowhammer breaks the assumptions about memory isolation in operating systems, the bug was quickly found to have severe security consequences. Several proof-of-concept exploits appeared shortly after the initial findings. First, randomized and double-sided rowhammer attacks were used to gain kernel privileges by breaking out of a virtualized environment[21]. Later, deterministic and reliable rowhammer attacks were used in an exploit for breaking memory isolation of virtual machines[18] as well as in an Android-based kernel exploit[27].

Similar to rowhammer, Tang et al. [25] show how unprivileged dynamic voltage control instructions can be used to induce bitflip faults in the secure zone of Android devices. They leverage this to induce faults in the implementation of RSA to lift private keys, thus gaining the ability to self-sign malicious applications on the device.

Hansen et al.[9] propose a formal analysis for verification of fault tolerance of programs by enforcing *blue/green separation*, which separates the computation of critical values. They prove how blue/green separation provides fault tolerance under data and flag SEU fault models. However, since their formalisation requires special instructions for atomically comparing multiple values for equality, they provide gadgets - small programs that semantically provides the comparison needed. Furthermore, they leverage statistical model checking to quantify the risk of faults in control register and instruction encoding SEUs. They show how blue/green separation still provides meaningful mitigation under these aggressive fault models as well. Their analysis targets a subset of the ARM language, which we have based our target language on.

Munkby and Schupp[16] suggest that conventional black-box fault injection methods of transient faults provide only limited insight into the behaviour of programs in the presence of faults. Risk assessments using such techniques are therefore not useful for general comparison with related applications. They propose to group fault injections by the usage pattern of variables, and claim that this is a good predictor for test and fault tolerance mitigation prioritisation. They classify variables according to their usage in the program: floating-point values, memory offsets, or control flow guards. They develop a type system for decorating variables by the above usage patterns, and show that variables used for floating-point operations are especially vulnerable to faults (for their evaluation program). Their idea of usage-pattern grouping is related to our idea of fault equivalence. However, fault equivalence classes are less approximative than theirs, since we group faults by the actual effect on program behaviour, whereas they apply usage-patterns as a heuristic of how susceptible variables are to faults.

Meola and Walker[13] develop a logic for reasoning about fault tolerance redundancies based on separation logic. Similar to us, they model faults by an explicit pseudo-instruction. Interestingly, they informally note that it is sufficient to insert faults for a variable immediately before the variable's value is read. This observation is directly related to our notion of fault equivalence, as it excludes many equivalent faults.

Perry and Walker[17] analyse the problem of maintaining control flow integrity in the presence of bitflip faults. They formalise the operational semantics of an assembly language type system to model invalid control flow. They leverage special intent registers, which are copies of the intended control flow targets. They use these in order to check whether the integrity of jump targets is preserved.

While most work on transient faults is focused on safety and security critical systems, Feng et al.[7] forecast that, as hardware dimensions are scaled down, the frequency of transient faults is scaled up. Because of this, they argue that handling transient faults in the context of commodity hardware will in time become a necessity. Since non-commercial users are not likely to willingly sacrifice much performance for fault tolerance, Feng et al. focus on "fault tolerance on the cheap" and provide a simple, yet effective heuristic for minimal instruction duplication. Their duplication scheme is based on classification of instructions according to the likelihood of a fault generating a symptom and whether the instruction can cause a "user-visible" effect. However, they do not differentiate between the types of instructions that may have a higher probability of masking faults.

Shoshitaishvili et al.[22] present a binary analysis framework that can be used to analyse the firmware running on embedded devices. Their analysis utilises symbolic execution and program slicing in order to detect authentication bypass vulnerabilities (*backdoors*) in firmware. They show how their analysis is able to detect exploitable backdoors in some commercially available devices. They use a so-called *security policy* in order to detect security critical code. This is the inspiration behind the *security assertions* that we use to detect illegal control flow in our analysis.

Other work has also focused on analysing program binaries. This often involves converting the binary data to an intermediate language, which allows the analysis tools to analyse binary programs from different architectures. This is what is done in analysis frameworks like *angr*[22, 23, 24], which also contains a symbolic execution engine. We choose to create our own analysis and symbolic execution engine, as this allows us to work with our target language without transforming it to an intermediate language. By letting the analysis work directly on our ARM based language, we are able to analyse the programs on a very low level and track the effects of individual instructions.

Reis et al.[19] present a software-based technique for achieving fault tolerance through redundancy. Their technique duplicates a program's instructions and inserts comparison instructions to compare important values at certain points in the program. This is done by duplicating the values of a program and thereby doing certain computations twice. The values are then checked for equivalence before they are used for critical operations. By doing this, they can detect if a fault has happened in any of the values and react to this. We will be experimenting with a similar duplication scheme in order to test our analysis.

3 TinyARM with Faults

This section is based on our previous work[2]. It contains an updated formalisation of the target language, which now uses 16-bit values instead of 32-bit. Using a smaller word size simplifies formalisation and examples, but the analysis does not depend on it in any other way. Furthermore, the syntax and semantics of a fault instruction have been added.

The formalisation of TinyARM is based on the formalisation in [9] and the ARM language reference manual[3]. The ARM language is a popular instruction set for embedded devices, which in turn are the systems that are typically exposed to environmental factors causing SEUs, or are available for adversaries to intentionally induce faults by techniques like rowhammer.

TinyARM is a subset of the ARM assembly language and features move, load, comparison, branching, and arithmetic instructions[3]. Note that TinyARM does not contain any instructions for conditional execution, which instead is realised through parameterized condition codes on every instruction. To formally model faults in a TinyARM program, we also extend the TinyARM language to include an explicit fault instruction.

Additionally, we assume the following about the programs we analyse:

- No loops / recursion.
- No function calls.
- No indirect jumps.

Although these assumptions impose significant limitations on the class of programs that can be expressed, it is done to simplify later analysis. We discuss issues and ideas for extending the analysis to loops in Section 11. While we will not consider indirect jumps, they can be resolved by existing techniques based on value-set analysis[4, 23, 9].

Before we introduce the syntax and semantics of TinyARM, we formalise the values that are used in TinyARM.

Let $\mathbb{B}$ be the set of binary values:

$$\mathbb{B} = \{0, 1\}$$

we consider 16-bit integer values encoded as binary numbers:

$$\mathbb{B}^{16} = \{0, ..., 15\} \rightarrow \mathbb{B}$$

Val is the set of possible values, encoded as 16-bit integers:

$$\mathit{Val} = \mathbb{B}^{16}$$

Addresses are also encoded as 16-bit integer values:

$$\mathit{Addr} = \mathbb{B}^{16}$$

For simplicity of the following formalisations and examples, we will usually write values as either decimal or hexadecimal values instead of full 16-bit words. When we write values as binary, we use the subscript $_2$ to denote that it is a binary value, e.g. $11_2 = 3$.

The ARM assembly language contains 16 registers that are used for different purposes[3]. Registers `r0` - `r12` are general purpose registers with no special hardware purpose. Registers `r13` and `r14` are normally used as the stack pointer and link register. These are function related registers and are therefore not included in the TinyARM formalisation. Finally, register `r15` is the program counter which keeps track of the 'current' instruction.

We denote the set of general purpose registers as

$$\mathit{DataReg} = \{\texttt{r0}, ..., \texttt{r12}\}$$

the set of control registers (only the program counter) as

$$\mathit{CtrlReg} = \{\texttt{r}_{\texttt{pc}}\}$$

and the set of all registers as

$$\mathit{Reg} = \mathit{DataReg} \cup \mathit{CtrlReg}$$

All registers and addresses contain values, so the register and memory environments are mappings from registers/addresses to values:

$$\mathit{Registers} = \mathit{Reg} \to \mathit{Val}$$

$$\mathit{Memory} = \mathit{Addr} \to \mathit{Val}$$

We denote updating a register, memory, or flag environment member x with the value v as $x \mapsto v$, e.g. $R[x \mapsto v]$ denotes a new register environment where register x maps to value v.

The ARM assembly language has four flags that are used to indicate conditions set by previous instructions[3]. The conditional flags keep track of negative, zero, carry, and overflow results. Each of these flags are set to a binary value:

$$\mathit{Flag} = \{f_N, f_Z, f_C, f_V\} \qquad \mathit{Flags} = \mathit{Flag} \to \mathbb{B}$$

In TinyARM each instruction is parameterized by a conditional code, which indicates whether the instruction should be executed based on the state of *Flags*. We denote the set of condition codes as:

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

The semantics for the condition codes are given by the *cond* function:

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

where $\chi \in \mathsf{ConditionCode}$.

The condition code `AL` is used for unconditional execution of an instruction. In the following examples and formalisation, we normally omit the condition code `AL`, meaning that any instruction without a condition code will be unconditionally executed. If the condition of an instruction does not evaluate to true, the instruction will be treated as a `NOP` instruction, which will just increase the program counter.

The semantics for flag updates under addition is given by the $\mathit{flags}_{\texttt{ADD}}$ function. For brevity, we show only the function for addition, as the rules for comparison, subtraction and multiplication are similar. The effects of comparison, subtraction, and multiplication instructions on the conditional flags are shown in the functions in Appendix A.

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_N) = \begin{cases} 1 & \text{if } v_1 + v_2 \geq 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_Z) = \begin{cases} 1 & \text{if } v_1 + v_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_C) = \begin{cases} 1 & \text{if } v_1 + v_2 > 2^{16} - 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_V) = \begin{cases} 1 & \text{if } v_1, v_2 < 2^{15} \wedge (v_1 + v_2 \geq 2^{15}) \\ 1 & \text{if } v_1, v_2 \geq 2^{15} \wedge (v_1 + v_2 < 2^{15}) \\ 0 & \text{otherwise} \end{cases}$$

In the above, v_1 and v_2 are the values of the registers that are used as operands of an addition instruction, while F is the current flags environment. The reason we also have F as input is that the `MULS` instruction only updates some of the conditional flags, see Appendix A. The above functions show the effect that an addition can have on the conditional flags. We can see that the negative flag, f_N, is set if the addition of two values results in a negative value. The zero flag, f_Z, is set if the result of the addition is zero. The carry flag, f_C, is set if more than 16 bits are required to store the result of the addition. Finally, the overflow flag, f_V, is set if the addition results in a signed overflow.

We now formally define the syntax of the core TinyARM language.

Definition 1 (Syntax of TinyARM)
The set of core TinyARM instructions is defined by the following abstract syntax:

$$\begin{aligned} \texttt{Instr} := {} & \texttt{MOV}\chi\, r_1,\ v \\ \mid {} & \texttt{MOV}\chi\, r_1,\ r_2 \\ \mid {} & \texttt{LDR}\chi\, r_1,\ a \\ \mid {} & \texttt{CMP}\chi\, r_1,\ r_2 \\ \mid {} & \texttt{OP}\chi\, r_1,\ r_2,\ r_3 \\ \mid {} & \texttt{OPS}\chi\, r_1,\ r_2,\ r_3 \\ \mid {} & \texttt{B}\chi\, a \end{aligned}$$

where $r_1, r_2, r_3 \in \mathit{DataReg}, v \in \mathit{Val}, a \in \mathit{Addr}, \chi \in \mathsf{ConditionCode}$, and $\texttt{OP} \in \{\texttt{ADD}, \texttt{SUB}, \texttt{MUL}\}$.

Note that the comparison and arithmetic instructions can only have registers as operands, unlike the ARM assembly language where some of the operands may be immediate values. However, we can still express the same programs since we are able to move the value into a register (using $\texttt{MOV}\chi\ r_1,\ v$) prior to the comparison or arithmetic instructions. In the case of branch instructions, we often write branch targets using labels in order to make programs easier to follow.

In order to model that faults may occur in programs, we extend the core TinyARM language with an explicit fault instruction as follows:

$$\begin{aligned}\texttt{InstrF} := {} & \texttt{Instr} \\ & \mid \texttt{FAULT}\ r,\ i\end{aligned}$$

where $r \in \mathit{DataReg}$ and $0 \leq i \leq 15$.

The advantage of modelling faults as explicit instructions is twofold. First, it simplifies our formalisation of fault equivalence in Section 5, since it allows us to reason about the exact program point, register, and bit position that the fault affects. Secondly, it enables us to check the possible vulnerabilities created by each fault in the vulnerability analysis step in a controlled manner.

With the syntax defined, we now introduce the notion of a program. A TinyARM program is a partial mapping from addresses to instructions:

$$\mathit{Program} = \mathit{Addr} \rightharpoonup \texttt{InstrF}$$

Since faults can occur any time during execution, fault instructions for every combination of register and bit position can be inserted at every program point. Although many of the fault instructions have no effect on the registers used in the program, it represents the possibility of such faults occurring. However, for vulnerability analysis, it is sufficient to analyse only the faults that can affect the behaviour of the program execution. In Section 8 we examine how to determine the relevant subset of faults for a given program.

Example 2 shows TinyARM assembly of the alarm controller program from Listing 1 in Section 1.

Example 2
Recall the alarm controller program from Listing 1. A similar program in TinyARM is shown in Listing 2. This program has combined the handling of error values and safe values. A couple of fault instructions have also been added to the program.

This program starts by declaring the values that it needs during execution. It then checks the sensor input (from Line 1) against the different bounds in the `CMP` instructions. Depending on the sensor value, the program will either sound the alarm or do something else. These actions are represented by the branch targets `alarm` and `noalarm`.

Two fault instructions have also been added to the program in Line 2 and Line 8. If the fault in Line 2 occurs, the value from the sensor will have its least significant bit flipped. This corrupts the input and could influence the program's execution. The fault in Line 8 is more dangerous.

```
LDR r0, 0xFEED     ; Read sensor.
FAULT r0, 0        ; Potential fault on r0.
MOV r1, 0          ; Min. error bound.
MOV r2, 10000      ; Max. error bound.
MOV r3, 2000       ; Dangerous temp.
CMP r0, r1         ; Check min.
BLT noalarm
FAULT r2, 13       ; Potential fault on r2.
CMP r0, r2         ; Check max.
BGT noalarm
CMP r0, r3         ; Check danger.
BLT noalarm
alarm:
...                ; Sound the alarm.
B exit
noalarm:
...                ; Do not sound the alarm.
exit:
```

Listing 2: Alarm controller program in TinyARM.

This fault flips the upper bound that checks for error values from 10000 to 1808. This means that any sensor value above 1808 will not sound the alarm, since they are considered sensor errors. In this case the program can no longer sound the alarm, since the sensor value would need to be less than 1808 and greater than 2000 at the same time in order to reach the alarm code.

The above example shows the main idea behind fault instructions. Note that the presence of a fault instruction does not imply that the fault occurs during execution of the program. Whether a fault occurs is determined by the following definition of a fault.

Definition 2 (Faults)
Let $P \in \mathit{Program}$ then the faults occurring in P is a function

$$\mathit{Faults} = \mathit{Addr} \to \{\mathit{true}, \mathit{false}\}$$

that maps addresses of a program to true or false. If the instruction at the given address is a `FAULT` instruction, a *Faults* function controls whether the instruction is executed or skipped. For any other instruction the *Faults* function has no effect. We will use f to denote some arbitrary fault function.

We use fault functions to control which fault instructions are enabled and which are not. As we shall see in the fault semantics, fault instructions that are not enabled are simply treated as `NOP` instructions. When we discuss faults enabled at specific program points, we let f_a denote a fault function that maps to true for address a and false for any other address, that is:

$$f_a(x) = \begin{cases} \mathit{true} & \text{if } x = a \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

The fault function idea can easily extended to fault models where multiple faults can occur during program execution, by allowing multiple faults to be enabled. Under our SEU fault model, however, we assume that exactly one fault instruction is enabled for any given fault function. Therefore the occurrence of different faults on different program executions are represented as separate fault functions.

Before we formally define the semantics of TinyARM, we define the relation over binary values that differ in exactly one given bit position. That is, we define the binary relation over binary values that has a Hamming distance of exactly one[9], on a specific bit position.

Definition 3 (1-Hamming Distance Relation)
Let $b_1, b_2 \in \mathbb{B}^{16}$ and $i, j \in \{0, ..., 15\}$ define the 1-Hamming distance relation ($\equiv_1^i$):

$$b_1 \equiv_1^i b_2 \quad \text{if and only if} \quad \forall j : b_1(j) \neq b_2(j) \iff i = j$$

The program, program counter, state of the registers, memory, and flags together form the state of the program execution called a *configuration*. Formally, we define configurations as

$$\mathit{Conf} = \mathit{Program} \times \mathit{Addr} \times \mathit{Registers} \times \mathit{Memory} \times \mathit{Flags}$$

Before we formalise the semantics of TinyARM, we define the transitions between configurations. More formally, we define the binary transition relation

$$\Longrightarrow \; \subseteq \mathit{Conf} \times \mathit{Conf}$$

For simplicity, we adopt the notation $C \Longrightarrow C'$ for denoting $(C, C') \in \Longrightarrow$ for $C, C' \in \mathit{Conf}$. We let $\Longrightarrow^*$ denote the reflexive and transitive closure of $\Longrightarrow$, and $\Longrightarrow^n$ denote a reduction sequence of length n.

A TinyARM program forms a transition system according to the semantic rules shown in Figure 1. In the semantics, a configuration can transition to another configuration if any rule is satisfied. As the configurations process through the rules they execute the program, until the program counter goes out of range (i.e. until the last instruction has been executed).

The semantic rules follows the semantics of ARM closely[3]. All instructions are executed only if the conditions for the condition code are satisfied. Any flag-setting instruction updates the state of the *Flags* environment according to the *flags* function.

In the semantic rules for the arithmetic operations, the *op* symbol is used to indicate the corresponding arithmetic symbol (e.g. for `ADD`, *op* will be $+$).

The `NOP` rule is applied whenever the condition check is not satisfied. As previously mentioned, this has the effect of the instruction not being executed, and simply increments the program counter to the next instruction.

Finally, if the instruction is a fault instruction, one of two cases can happen. If the fault is enabled under the given fault function f (note that the configuration for faults are parameterized by a global fault f) the target register r is updated to a new value. The new value is determined by the target bit position i of the fault instruction and Definition 3. Essentially, the i^{th} bit of the value of r is changed to its complement. As mentioned previously, if the fault is not enabled under the fault function f, execution simply treats the fault instruction as a `NOP` and proceeds to the next instruction with the configuration intact.

$$[\texttt{MOV-VAL}]\ \frac{P(PC) = \texttt{MOV}\chi\, r,\ v \qquad \mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r \mapsto v], M, F\rangle}$$

$$[\texttt{MOV-REG}]\ \frac{P(PC) = \texttt{MOV}\chi\, r_1,\ r_2 \qquad \mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r_1 \mapsto R(r_2)], M, F\rangle}$$

$$[\texttt{LDR}]\ \frac{P(PC) = \texttt{LDR}\chi\, r,\ a \qquad \mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r \mapsto M(a)], M, F\rangle}$$

$$[\texttt{CMP}]\ \frac{P(PC) = \texttt{CMP}\chi\, r_1,\ r_2 \qquad \mathit{cond}(\chi, F) \qquad F' = \mathit{flags}_{\texttt{CMP}}(R(r_1),\ R(r_2),\ F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R, M, F'\rangle}$$

$$[\texttt{OP}]\ \frac{P(PC) = \texttt{OP}\chi\, r_1,\ r_2,\ r_3 \qquad \mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r_1 \mapsto R(r_2)\ \mathit{op}\ R(r_3)], M, F\rangle}$$

$$[\texttt{OPS}]\ \frac{P(PC) = \texttt{OPS}\chi\, r_1,\ r_2,\ r_3 \qquad \mathit{cond}(\chi, F) \qquad F' = \mathit{flags}_{\texttt{OP}}(R(r_1),\ R(r_2),\ F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r_1 \mapsto R(r_2)\ \mathit{op}\ R(r_3)], M, F'\rangle}$$

$$[\texttt{B}]\ \frac{P(PC) = \texttt{B}\chi\, a \qquad \mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, a, R, M, F\rangle}$$

$$[\texttt{NOP}]\ \frac{P(PC) = \texttt{Instr}\chi \qquad \neg\mathit{cond}(\chi, F)}{\langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R, M, F\rangle}$$

$$[\texttt{FAULT}_{\texttt{true}}]\ \frac{P(PC) = \texttt{FAULT}\, r,\ i \qquad f(PC) = \texttt{true} \qquad v \equiv_1^i R(r)}{f \vdash \langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R[r \mapsto v], M, F\rangle}$$

$$[\texttt{FAULT}_{\texttt{false}}]\ \frac{P(PC) = \texttt{FAULT}\, r,\ i \qquad f(PC) = \texttt{false}}{f \vdash \langle P, PC, R, M, F\rangle \Longrightarrow \langle P, PC+1, R, M, F\rangle}$$

Figure 1: Formal semantics of TinyARM with faults.

4 Symbolic Execution

This section is based on our previous work[2]. It contains a modified way of collecting flags and conditions during symbolic execution. Furthermore, symbolic execution semantics for the new fault instruction and our security assertion have been added.

In this section we formalise the analysis for collecting the path conditions that must be satisfied in relevant program executions by *symbolic execution*, as well as how we utilize special assertions to detect vulnerable program traces caused by faults.

Symbolic execution is a technique for analysing the behaviour of programs by abstracting execution on many different inputs at the same time. The main idea is that a specific program trace can be encoded as the logical conjunction of the conditions that must be satisfied for a program point to be reached. The behaviour of a program can therefore be expressed as a set of conditions on registers, memory, and flags used in the control flow of an execution. This means that the primary objective of symbolic execution is to construct a logic formula that expresses the conditions on resources used in conditional instructions. Since such logic formulas encode the behaviour of the program, we have effectively reduced the problem of reasoning about program behaviour to the domain of logic. We can obtain solutions to satisfy the paths by solving the formulas with an SMT solver.

We formalize symbolic execution as operational semantics as Schwartz et al.[20]. While the formalisation is unambiguous and provides a straightforward implementation of the analysis, it should be noted that the complexity of this implementation is not efficient since it is exponential in the number of branches, in the number of path conditions and has an exponential-size formula for each branch[20]. However, we let the complexity of the analysis be an implementation detail, but note that there exists more efficient methods for calculating the path condition, e.g. clever caching of sub formulas or using the weakest precondition[20].

We extend the core operational semantics of TinyARM in order to collect path conditions in a special condition environment during execution. Since TinyARM conditional execution depends on the value of the conditional flags, the conditions are generated based on the flag semantics.

When we do not know the concrete value of the register during symbolic execution, we let the content of a register be symbolic expressions. To model these, we define an infinite set of symbols that we can assign to registers:

$$Symbol = \{ \ s_i \mid i \in \mathbb{N} \ \}$$

We also define the function $NewSymbol$ which returns a new fresh symbol each time it is used.

Furthermore, we define the following expression language, which is used to describe register values during symbolic execution.

$$\rho := \ v \mid s \mid \rho + \rho \mid \rho - \rho \mid \rho * \rho$$

where $v \in \ Val$ and $s \in Symbol.$

In order to handle the new register values, we define a new $Registers$ function. This maps a program point to the registers and their values at that program point.

$$Registers^{+} = Addr \rightarrow DataReg \rightarrow \rho$$

The reason for recording the values of the registers at all program points is that the security assertions can refer to the concrete values of registers at certain program points. Essentially, the new register environments is a collection of the previous definition of register environments.

We also define a new $Memory$ function which is used during symbolic execution.

$$Memory^{+} = Addr \rightarrow \ Val \cup Symbol$$

The $Memory^{+}$ function is initialized with unique symbols on every address. As a consequence, the memory is by default fully symbolic. We can still supply input by overriding a given address with a specific value prior to running symbolic execution, e.g. $M[\texttt{0xFEED} \mapsto 42]$.

As we shall see in the semantic rules for symbolic execution, updating register and flag environments now includes copying the the mappings from previous program points. In Example 3 we clarify how updates work on the symbolic version of the register environments.

Example 3 (Updating Registers)
Suppose that we have the register environment R where

$$R(1)(\texttt{r0}) = 2,\ R(1)(\texttt{r1}) = 42$$

and we want to update `r1` to the immediate value 256 in the next program point (2). We use the following update statement to obtain the updated register environment:

$$R' = R[2 \mapsto R(1)[\texttt{r1} \mapsto 256]]$$

and we now have that

$$R'(1)(\texttt{r0}) = 2,\ R'(1)(\texttt{r1}) = 42,\ R'(2)(\texttt{r0}) = 2,\ R'(2)(\texttt{r1}) = 256$$

Essentially, the values are "copied forward" and the value at the new program point is updated. Updates on flag and memory environments are conceptually similar.

We note that while the updates in our semantics are not standard, it is inspired by copying forward values in analysis specification with flow-logic [9], except that it is more procedural than declarative.

Since we can no longer evaluate register values when updating the conditional flags, we introduce the boolean expression language β. This is used to represent the path conditions, and later on the full logic formula to be solved.

$$\begin{array}{ll} \beta_X := x_1 = x_2 & \mid x_1 \neq x_2 \\ \quad\ \mid x_1 < x_2 & \mid x_1 \leq x_2 \\ \quad\ \mid x_1 > x_2 & \mid x_1 \geq x_2 \\ \quad\ \mid \neg\beta_X & \mid \beta_X \vee \beta_X \\ \quad\ \mid \beta_X \wedge \beta_X & \end{array}$$

where $x_1, x_2 \in X$.

This expression language is parameterized with X, which allows it to be used in conjunction with other expression languages, e.g. β_ρ is used for path condition expressions while β_α is used for security assertions. This is done to ensure that only security assertions may refer to values of registers, flags and memory at previous program points.

The comparison and arithmetic instructions of TinyARM still updates the value of the conditional flags during symbolic execution, but since we can no longer evaluate the registers, the effect of the update is different. The functions for updating the flags are found in Appendix B. These new update functions simply set the value of the flags to be the boolean expression that describes whether they are true or false.

We now define a new *Flags* function that provides the expressions of the conditional flags.

$$\mathit{Flags}^+ = \mathit{Addr} \rightarrow \mathit{Flag} \rightarrow \beta_\rho$$

The Flags^+ function also stores the previous values of the flags, since these are used later in the analysis.

In order to generate conditions from conditional flags, we define the *cons* function that maps a condition code χ and the state of the flags to expressions that must be satisfied for χ to hold:

$$
cons(\chi, (f_N, f_Z, f_C, f_V)) = \begin{cases}
f_Z & \text{if } \chi = \texttt{EQ} \\
\neg f_Z & \text{if } \chi = \texttt{NE} \\
f_C & \text{if } \chi = \texttt{CS} \\
\neg f_C & \text{if } \chi = \texttt{CC} \\
f_N & \text{if } \chi = \texttt{MI} \\
\neg f_N & \text{if } \chi = \texttt{PL} \\
f_V & \text{if } \chi = \texttt{VS} \\
\neg f_V & \text{if } \chi = \texttt{VC} \\
f_C \wedge \neg f_Z & \text{if } \chi = \texttt{HI} \\
\neg f_C \vee f_Z & \text{if } \chi = \texttt{LS} \\
f_N = f_V & \text{if } \chi = \texttt{GE} \\
f_N \neq f_V & \text{if } \chi = \texttt{LT} \\
\neg f_Z \wedge (f_N = f_V) & \text{if } \chi = \texttt{GT} \\
f_Z \vee (f_N \neq f_V) & \text{if } \chi = \texttt{LE} \\
\texttt{true} & \text{if } \chi = \texttt{AL}
\end{cases}
$$

where $\chi \in \mathsf{ConditionCode}$.

Next, we extend the core TinyARM syntax to include security assertions used for identifying whether a fault caused an illegal change in control flow. Before we do this, however, we introduce the expressions that are used by the assertions:

$$\alpha = \ v \mid r@a$$

where $v \in \mathit{Val}$, $r \in \mathit{DataReg}$, and $a \in \mathit{Addr}$. The expression $r@a$ refers to the value of the register r at the address a, which can be retrieved from the $\mathit{Registers}^+$ environment.

We now formally extend the TinyARM language to include the security assertion.

Definition 4 (Syntax of TinyARM Assert)
TinyARM is extended to include the security assertion:

$$
\begin{aligned}
\texttt{InstrA} := {} & \texttt{InstrF} \\
\mid {} & \texttt{ASSERT}\ c
\end{aligned}
$$

where $c \in \beta_\alpha$.

A security assertion is written in the program in order to mark the critical program point for the analysis. The critical program point could be instructions that should only be executed under certain privileges, or code that triggers safety measures like in the alarm controller program.

We imagine that security assertions are manually specified by the developer since assertions in safety and security critical systems already should contain various assertion statements, or automatically generated by some previously applied program analysis.

Since we updated the TinyARM syntax to include the security assertion, we need to update the *Program* function to work with this instruction. We do this by redefining the old environment:

$$\mathit{Program} = \mathit{Addr} \rightharpoonup \texttt{InstrA}$$

The conditions collected by the symbolic execution are handled by the *Conditions* environment.

$$\mathit{Conditions} = \mathit{Addr} \to \beta_\rho$$

Finally, we define the configurations that are used for symbolic execution.

$$\mathit{Conf}^+ = \mathit{Program} \times \mathit{Addr} \times \mathit{Registers}^+ \times \mathit{Memory}^+ \times \mathit{Flags}^+ \times \mathit{Conditions}$$

Before we present the operational semantics for symbolic execution, we extend the transition relation $\Longrightarrow$ for the new type of symbolic transitions. Formally, we define the binary symbolic transition relation

$$\overset{s}{\Longrightarrow} \subseteq \mathit{Conf}^+ \times \mathit{Conf}^+$$

For simplicity, we adopt the notation $C \overset{s}{\Longrightarrow} C'$ for denoting $(C, C') \in \overset{s}{\Longrightarrow}$. We let $\overset{s}{\Longrightarrow}^*$ denote the reflexive and transitive closure of $\overset{s}{\Longrightarrow}$, and $\overset{s}{\Longrightarrow}^n$ denote a reduction sequence of length n.

We also define the notion of a program trace.

Definition 5
A trace is a sequence $(c, ..., c') \in (\mathit{Conf}^+)^*$ such that $c \overset{s}{\Longrightarrow}^* c'$. Let $\mathit{Traces}(c \overset{s}{\Longrightarrow}^* c')$ denote the set of all traces from c to c'.

Instead of writing $T \in \mathit{Traces}(c \overset{s}{\Longrightarrow}^* c')$, we adopt the notation $T \in c \overset{s}{\Longrightarrow}^* c'$ to indicate T is a trace from c to c'.

The operational semantic rules specifying the symbolic execution analysis are shown in Figure 2. These rules can be used to symbolically execute TinyARM programs, like the rules in Figure 1, while collecting conditions that reason about the value of any symbol.

While the semantics for symbolic execution are similar to the core semantics in Section 3, there are a few important differences.

One principal difference is that conditions are collected and propagated through the execution. Every rule generates conditions according to the semantics of the *cons* function and conjoins them to the condition environment. Since any instruction may be executed conditionally depending on the condition code, we have that any instruction may potentially generate conditions. For simplicity in the following examples and formalisation, we omit any trivial tautology clauses, e.g. $\texttt{true} \land \texttt{true} \land ...$, resulting from the conditions generated by instructions with the `AL` condition code.

Once the symbolic execution semantics have finished collecting conditions, we can use an *SMT solver* in order to solve the conditions. By solving the conditions, we gain insight into the program behaviour and information about possible symbol values. We are using the Z3 Theorem Prover[6] for solving the conditions since it supports quantified bit vectors which provides a straightforward way to encode the conditions generated by Definition 3 in the semantic rule for fault instructions.

$$[\texttt{MOV-VAL}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{MOV}\chi\, r,\ v \\ R' = R[PC+1 \mapsto R(PC)[r \mapsto v]] \\ F' = F[PC+1 \mapsto F(PC)] \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{MOV-REG}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{MOV}\chi\, r_1,\ r_2 \\ R' = R[PC+1 \mapsto R(PC)[r_1 \mapsto R(PC)(r_2)]] \\ F' = F[PC+1 \mapsto F(PC)] \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{LDR}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{LDR}\chi\, r,\ a \\ R' = R[PC+1 \mapsto R(PC)[r \mapsto M(a)]] \\ F' = F[PC+1 \mapsto F(PC)] \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{CMP}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{CMP}\chi\, r_1,\ r_2 \\ R' = R[PC+1 \mapsto R(PC)] \\ F' = F[PC+1 \mapsto \mathit{flagsSe}_{\texttt{CMP}}(R(r_{pc})(r_1),\ R(r_{pc})(r_2),\ F) \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{OP}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{OP}\chi\, r_1,\ r_2,\ r_3 \\ R' = R[PC+1 \mapsto R(PC)[r_1 \mapsto R(PC)(r_2)\ \mathit{op}\ R(PC)(r_3)]] \\ F' = F[PC+1 \mapsto F(PC)] \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{OPS}]\ \dfrac{\begin{array}{l} P(PC) = \texttt{OPS}\chi\, r_1,\ r_2,\ r_3 \\ R' = R[PC+1 \mapsto R(PC)[r_1 \mapsto R(PC)(r_2)\ \mathit{op}\ R(PC)(r_3)]] \\ F' = F[PC+1 \mapsto \mathit{flagsSe}_{\texttt{OP}}(R(r_{pc})(r_1),\ R(r_{pc})(r_2),\ F)] \\ C' = C[PC+1 \mapsto C(PC) \wedge \mathit{cons}(\chi, F)] \end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{B}]\ \frac{\begin{array}{l}P(PC) = \texttt{B}\chi\ a\\ R' = R[a \mapsto R(PC)]\\ F' = F[a \mapsto F(PC)]\\ C' = C[a \mapsto C(PC) \land \mathit{cons}(\chi, F)]\end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, a, R', M, F', C'\rangle}$$

$$[\texttt{FALSE}]\ \frac{\begin{array}{l}P(PC) = \texttt{Instr}\chi\\ R' = R[PC+1 \mapsto R(PC)]\\ F' = F[PC+1 \mapsto F(PC)]\\ C' = C[PC+1 \mapsto C(PC) \land \neg\mathit{cons}(\chi, F)]\end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{FAULT}_{\texttt{true}}]\ \frac{\begin{array}{l}P(PC) = \texttt{FAULT}\ r,\ i\\ f(PC) = \texttt{true}\\ s = \mathit{NewSymbol}\\ R' = R[PC+1 \mapsto R(PC)[r \mapsto s]]\\ F' = F[PC+1 \mapsto F(PC)]\\ C' = C[PC+1 \mapsto C(PC) \land s \equiv_1^i R(PC)(r)]\end{array}}{f \vdash \langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{FAULT}_{\texttt{false}}]\ \frac{\begin{array}{l}P(PC) = \texttt{FAULT}\ r,\ i\\ f(PC) = \texttt{false}\\ R' = R[PC+1 \mapsto R(PC)]\\ F' = F[PC+1 \mapsto F(PC)]\\ C' = C[PC+1 \mapsto C(PC)]\end{array}}{f \vdash \langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

$$[\texttt{ASSERT}]\ \frac{\begin{array}{l}P(PC) = \texttt{ASSERT}\ c\\ R' = R[PC+1 \mapsto R(PC)]\\ F' = F[PC+1 \mapsto F(PC)]\\ C' = C[PC+1 \mapsto C(PC) \land c]\end{array}}{\langle P, PC, R, M, F, C\rangle \overset{s}{\Longrightarrow} \langle P, PC+1, R', M, F', C'\rangle}$$

Figure 2: Operational semantics for symbolic execution of TinyARM.

The LDR instruction also works different from the core semantics, since the memory can now be symbolic. This means that reading from memory can return either a value or a new symbol. This is one of two way that symbols are introduced. The other way is through the FAULT instruction, which generates a new symbol and constrains this to be 1-Hamming distance from the previous register value.

Another important difference is that since flags depend on symbolic values, they can no longer easily be evaluated to true or false. This means that we cannot readily determine which branch to select. Semantically, this means that there are two rules applicable in any conditional instruction since the semantic rule `FALSE` potentially matches any instruction. In operational semantics this implies a nondeterministic choice between the two possible rules, discarding the path not chosen. However, in our case the goal is to explore all possible execution paths to the target instruction.

The problem of deciding which branch to take in symbolic execution is known as the *path selection problem*[20]. There are several strategies for determining which branch to take first, including Depth-first search (DFS), random, concolic testing, and using heuristics. We adopt the DFS strategy, since the goal is to explore every path to the target. Normally, DFS has issues with nonterminating loops when guards contain symbolic expressions. In our case, a DFS strategy can be readily used for an exhaustive search since we assume that the programs do not contain loops.

The $\mathit{Registers}^{+}$ and Flags^{+} environments used by symbolic execution are also different from the $\mathit{Registers}$ and Flags environments used in the core TinyARM semantics. These new environments store the values of the registers and flags at all program points. This is done by copying the existing values forward in the execution. This means that once a register or flag has been set, it has the same value in the following program points, until it is redefined.

The newly added security assertion instruction (`ASSERT`) conjoins its conditions to the conditions environment and continues execution. This means that in order to reach the code following the security assertion, the conditions of the security assertion should be satisfied. During the analysis, however, we normally stop after the security assertion and checks the conditions to verify whether the analysed fault caused an illegal flow in the program. We can do this since we only have a single security assertion at a time, but the current semantics could be used to support multiple security assertions, in order to enable analysis of programs where security is broken only if two critical program points are reached.

We now give an example to illustrate how the symbolic execution works. Example 4 applies symbolic execution on the alarm controller program, show the conditions that are generated, and check whether they are satisfied.

Example 4 (Symbolic Execution)
Consider the alarm controller program in Listing 3. We have added a security assertion in Line 13 which states that the only time the alarm should not be sounded is if the temperature is outside the danger interval ($[2000 \,..\, 10000]$). This means that if a run reaches the `noalarm` code with a dangerous temperature, something must have changed the intended control flow.

Note that in this case the security assertion is designed to capture *false negatives* on the alarm. In order to analyse false positives on triggering the alarm, another assertion should be used.

The possible symbolic executions of the program can be represented as the graph in Figure 3. We have contracted the vertices of the tree. When symbolically executing the program we conceptually traverse the tree and collect the conditions on the edges. The conditions tell us what needs to be satisfied in order for the execution to reach a given program point.

```
1  LDR r0, 0xFEED    ; Read sensor.
2  MOV r1, 0         ; Min. error bound.
3  MOV r2, 10000     ; Max. error bound.
4  MOV r3, 2000      ; Dangerous temp.
5  FAULT r0, 15      ; Possible fault on r0.
6  CMP r0, r1        ; Check min.
7  BLT noalarm
8  CMP r0, r2        ; Check max.
9  BGT noalarm
10 CMP r0, r3        ; Check danger.
11 BLT noalarm
   alarm:
   ...
12 B exit
   noalarm:
13 ASSERT r0@1 < 2000 ∨ r0@1 > 10000
   ...
   exit:
```

Listing 3: Alarm controller used to show symbolic execution.

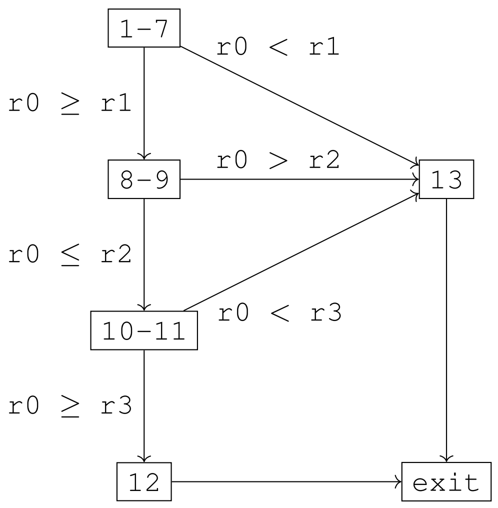


Figure 3: Symbolic execution of Listing 3.

Lets consider two runs of the program in which the sensor reads the value 8000, that is:

$$M(\texttt{0xFEED}) = 8000$$

In one of the runs the fault in instruction 5 is not enabled and in the other it is.

For the fault-free run it is not possible to reach the security assertion in the `noalarm` code (instruction 13). We can see this by looking at the conditions that are collected on the three possible paths from instruction 1 to instruction 13:

$$\begin{aligned}
&1 \to 13 && = 8000 < 0\\
&1 \to 8 \to 13 && = 8000 \geq 0 \wedge 8000 > 10000\\
&1 \to 8 \to 10 \to 13 && = 8000 \geq 0 \wedge 8000 \leq 10000 \wedge 8000 < 2000
\end{aligned}$$

None of these path conditions are satisfied, hence it is not possible to reach the `noalarm` code with the input 8000 in a fault-free run. It is only possible to reach the alarm code in instruction 12. This path will have the following condition:

$$8000 \geq 0 \wedge 8000 \leq 10000 \wedge 8000 \geq 2000$$

which is clearly satisfied.

If we look at the run with in the context of a the fault f_5, we see that it is possible to reach the security assertion in the `noalarm` code with the input 8000. When we reach the fault instruction in address 5, the fault occurs, and the value of `r0` will be changed such that the path from the first branch becomes:

$$s_0 \equiv_1^{15} 8000 \wedge s_0 < 0$$

which is satisfied since s_0 gets the value -24768.

This is not the intended program behaviour since an alarm should have been sounded with an input of 8000. In order to detect this, we use the security assertion by conjoining it to the path condition. Before we do this, however, we negate the security assertion. This is done to ensure that all possible assignments to the assertion's registers are considered when we solve the conditions. In other words, in order to prove that for all inputs the execution is safe, we prove that there exist no input such that the execution is unsafe.

For the example program, we know that `r0@1` can only be 8000, however in general, `r0@1` could be a symbol in which case we would have to consider several possible values. The conditions, after the negated assertion has been added, will look as follows:

$$s_0 \equiv_1^{15} 8000 \wedge s_0 < 0 \wedge \neg(8000 < 2000 \vee 8000 > 10000)$$

where $8000 =$ `r0@1`. The value of `r0@1` can be retrieved from the register environment, $\mathit{Registers}^+$. Using De Morgan's laws we can rewrite the conditions to:

$$s_0 \equiv_1^{15} 8000 \wedge s_0 < 0 \wedge 8000 \geq 2000 \wedge 8000 \leq 10000$$

which are satisfied. Since we negated the security assertion, a satisfied result means that the security assertion was broken and the fault was able to change the intended control flow of the program.

5 Fault Equivalence

Since a fault can occur at any point during execution, the number of possible fault instructions encoded in a program by our formalisation is substantial, spanning every register at every program point. Extending the types of faults to SEUs in memory, flags etc. result in even more faults encoded. However, many faults may cause the exact same effect on the behaviour of the program, despite having completely different characteristics, i.e. not targeting the same register or occurring at different times during execution.

Usually, we are not interested in the effect each fault has on the behaviour of a program, but instead on the overall possible effects that can be caused by faults. Even though the conventional fault injection methods combined with black-box simulation provides assessment of the fault tolerance of programs, it provides little to no insight into the nature of the faults causing the undesirable effects.

We propose to categorize faults into a set of *fault equivalence classes* conditioned on the effect they have on the behaviour of a program. Reasoning about equivalence classes of faults instead individual faults potentially reduces the complexity of our analysis and may provide much more useful insight into the fault tolerance of programs. Fault equivalence can also be a useful basis for further fault analyses.

We now show examples to illustrate the idea behind equivalent faults and to show cases where we expect faults to exhibit the same effect on the program. We then present formal definitions of fault equivalence relations and discuss how they capture that equivalent faults indeed cause the same effect on a program.

A simple example showcasing faults with the same effect on program behaviour is shown in Example 5.

Example 5
Consider the TinyARM program in Listing 4 with two fault instructions.

```
1 LDR r1, 0xFEED
2 FAULT r1, 0
3 MOV r2, 42
4 FAULT r1, 0
5 CMP r1, r2
6 ...
```

Listing 4: Equivalent faults on `r1` in the least significant bit position.

Observe that the two fault instructions affect the same definition of the same register (`r1` defined in instruction 1) on the same bit position (bit 0). Also, there are no instructions in the control-flow between the faults that affects `r1` nor uses `r1` in computations before the comparison in instruction 5. Regardless of previous execution, the effect of the faults f_2 and f_4 is identical. We argue that in general, there are equivalent faults at every program point in the path from the definition of a register to the next use of the same register.

Example 6
Listing 5 shows a TinyARM program with faults on different registers and different bit positions.

Note that unlike in Example 5, the two faults target different registers and bit positions. In order to decide if the faults are equivalent or not, we observe that in any execution, after instruction 2, `r1` maps to $1 = 1_2$ and `r2` maps to $2 = 10_2$. We now consider the cases when each fault occurs:

1. With f_3, the 0^{th} bit of `r1` is flipped and `r1` maps to $0_2 = 0$. Since we use `r1` for multiplication in instruction 5, we get that `r3` maps to 0 after instruction 5.

```
1  MOV r1, 1
2  MOV r2, 2
3  FAULT r1, 0
4  FAULT r2, 1
5  MULS r3, r1, r2
6  MOV r4, 0
7  CMP r3, r4
8  ...
```

Listing 5: Equivalent faults on different registers and different bit positions.

2. With f_4, the 1^{st} bit of `r2` is flipped, i.e. `r2` now maps to $0_2 = 0$. Since `r2` is used for multiplication in instruction 5, we get that `r3` maps to 0 after instruction 5.

By these two cases, we see that the effect of the faults differ on `r1` and `r2` but are identical for `r3`. Assuming that `r1` and `r2` are not live after instruction 5, we argue that the two faults are equivalent since they have the same effect on the live register `r3`.

Example 7
Consider the TinyARM program in Listing 6 which shows a program with two faults in distinct branches.

```
1  LDR r1, 0xFEED
2  LDR r2, 0xDEAD
3  LDR r3, 0x1234
4  CMP r1, r2
5  BGT gt
6  FAULT r3, i
7  MOV r4, 0
8  B check
9  gt:
10 FAULT r3, i
11 MOV r4, 1
12 check:
13 CMP r3, r4
14 BGT x
15 ...
```

Listing 6: Faults in distinct branches.

At first glance the two faults may seem equivalent since they affect the same register on the same bit position. However, suppose that we execute the program with the fault f_6. For all input that causes control to flow to the `gt` branch, we never propagate the effects of f_6, since the enabled fault instruction is never reached. Consequently, the flags after instruction 13 are consistent with fault-free execution. Now, consider the case when executing with fault f_{10}. For all input that causes control to flow to the `gt` branch, the `r3` register is corrupted, propagating the fault to the flags set by instruction 13. Therefore, the two faults may affect the program differently on the same input.

The above examples illustrate cases where we expect faults to be equivalent or distinct. We now propose definitions of fault equivalence.

The following definition states that two faults are simple equivalent whenever separately executing the program with the two faults yields the same configuration in the target program point.

Definition 6 (Simple Fault Equivalence)
Given a program $P \in \mathit{Program}$ and the address of the target security assertion $a \in \mathit{Addr}$, then two fault functions f, f' are simple equivalent on P ($f \equiv_f^P f'$) if and only if the following holds for any start configuration $\langle P, pc, R, M, F\rangle$:

$$f \vdash \langle P, pc, R, M, F\rangle \Longrightarrow^{*} \langle P, t, R', M', F'\rangle \text{ if and only if}$$
$$f' \vdash \langle P, pc, R, M, F\rangle \Longrightarrow^{*} \langle P, t, R', M', F'\rangle$$

Simple fault equivalence treats computation as a *black box* and defines equivalence purely in terms of whether the externally observable effects of two faults are the same.

Regarding the three example programs, we argue that the faults in Example 5 falls under simple fault equivalence, since `r1`, `r2`, and the flags will have the same final values regardless of the chosen fault.

The faults in Example 6 are not simple equivalent, since the activation of the first fault will result in `r1` $= 0$ and `r2` $= 2$, while the second fault results in `r1` $= 1$ and `r2` $= 0$. This observation leads us to that simple fault equivalence may be too strict on the final configuration.

Finally, the faults in Example 7 will have different final values for `r3` and are not simple equivalent.

This simple kind of equivalence can work in some scenarios. We may, however, want to enforce that the control flow does not deviate significantly when running the program with two different faults. While this may not be important in the current TinyARM language, it is important if the language is ever extended to include function calls. In this case an execution could change flow to call a function, and then reset the control flow changing registers before continuing. This would result in the same final state as executions that do not call the function. The function could, however, potentially influence something outside the program.

In order to relate equivalence to the control flow of a program, we introduce a new type of equivalence. The following equivalence definition states that two faults are equivalent whenever the flags are identical in every configuration of both fault runs for the same input.

Definition 7 (Control Fault Equivalence)
Given a program $P \in \mathit{Program}$ and the address of the target security assertion $a \in \mathit{Addr}$, then two fault functions f, f' are control equivalent on P ($f \equiv_f^P f'$) if and only if the following holds for any given start configuration $\langle P, pc_0, R_0, M_0, F_0\rangle$:

For all $i > 0$:

$$f \vdash \langle P, pc_0, R_0, M_0, F_0\rangle \Longrightarrow^{i} \langle P, pc_i, R_i, M_i, F_i\rangle \Longrightarrow^{*} \langle P, t, R_t, M_t, F_t\rangle \text{ if and only if}$$
$$f' \vdash \langle P, pc_0, R_0, M_0, F_0\rangle \Longrightarrow^{i} \langle P, pc_i, R_i', M_i', F_i\rangle \Longrightarrow^{*} \langle P, t, R_t', M_t', F_t\rangle$$

By enforcing that the flags at every instruction must be identical under both faults, the control flow must also be identical since the condition code semantics depend directly on the content of the flags. Furthermore, we abstract from any intermediate computation and the flow of corrupted values through general purpose registers, as long as the effect on the flags are the same. We now relate Definition 7 to the example programs.

The only instruction affecting the flags in Example 5 is the final `CMP` instruction. Since both `r1` and `r2` will have the same values under either fault function, the comparison will update the flags in the same way. This implies that the faults are control equivalent.

In Example 6 the flags will be updated twice: first by the `MULS` instruction and later by the `CMP` instruction. The result of the multiplication will be zero under both fault functions, and since the `MULS` instruction only updates the negative and zero flags, the flags under each fault function will be identical after the multiplication. The `CMP` instruction in the end will compare zero with zero under both fault functions, therefore its flags will be the same as well, hence the faults are equivalent.

The faults in Example 7 are not control equivalent. The reason for this is that the flags set by the second `CMP` instruction can be different, since only one of the faults are reached based on the chosen input.

Throughout the rest of the report, we consider only Control Fault Equivalence (Definition 7). For simplicity, we let $\equiv_f^P$ denote control equivalence. We do this since we are mostly concerned with the execution flow of programs under given faults. This allows us to reason about which faults, if any, can reach safety/security critical code in a program. Checking whether given faults are equivalent can be done in several ways. In Section 8 we will discuss how symbolic execution is used for deciding the equivalence of faults in our analysis prototype tool.

6 Analysis Overview

In this section we give an overview of the fault analysis and a short explanation of the steps involved in it. A flow chart of the complete analysis is found in Figure 4.

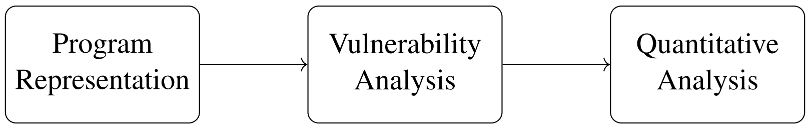


Figure 4: Fault analysis flow chart.

Program Representation
The analysis starts of by building several representations of the target program. These representations serve different purposes in the analysis. The main program representation used by the analysis is the *Control Flow Graph* (CFG), which shows the possible executions of the program. Next, several *dependency graphs* are generated. These capture different information about how instructions influence each other. The main purpose of the dependency graphs is to create a backwards slice of the program. This slice can be used to limit the parts of the program that should be analysed. The process of generating the various graphs as well as the backwards slice is explained in Section 7.

Vulnerability Analysis
The next step of the analysis is to select which faults we want to analyse. Instead of analysing every possible fault (in any register at any program point), we gradually reduce the number of faults that we need to consider. This is done to speed up the final simulation step of the analysis. The process of selecting faults to analyse is discussed in Section 8.

Quantitative Analysis
The last step of the analysis is to use statistical model checking to get various probabilistic results for the program. This allows the analysis to find out how vulnerable the program is to SEU faults by simulating a number of program executions. By doing this we can get the probability of a given fault breaking the safety/security of the program. The quantitative part of the analysis is discussed in Section 9.

7 Graphs and Backwards Slicing

This section is based on our previous work[2]. It contains short definitions of important program representations that we are reusing for this report.

In this section, we describe the different ways that we represent TinyARM programs in our analysis. These representations capture different information about the program and are known from many common static analyses. We also discuss our use of the backwards program slice and why it is useful.

The most important program representation that we are using is the *Control Flow Graph* (CFG). This is a graph that models the transition system formed by the semantics of TinyARM. The CFG is constructed by adding a vertex for each instruction in the target program, and adding edges to form the transition system of the program. This captures the possible ways that the program can be executed. Definition 8 formally defines control flow graphs.

Definition 8 (Control Flow Graph)
A Control Flow Graph (CFG), G, is a tuple $G = (V, E, \mathit{entry}, \mathit{exit})$ where

- $V \subseteq \mathit{Addr} \times \texttt{InstrA}$,
- $E \subseteq V \times V$,
- $\mathit{entry} \in V$ is the entry vertex, and
- $\mathit{exit} \in V$ is the exit vertex.

Our CFGs are formed in a very straightforward way from the program's instructions. The only instruction type that has more than one outgoing edge is the conditional branch, which can reach two different program points depending on the evaluation of its condition code. Other conditional instructions will reach the next program point regardless of their execution. It is common to group non-branching, sequentially executed instructions into basic blocks in CFGs to increase performance and decrease model complexity. However, we have mainly focused on smaller programs and as such do not use basic blocks. The complete transformation from a TinyARM program to a CFG is described in Appendix C.

A CFG represents all possible traces for a given program. In Definition 9 we define the notion of a path in a CFG, which is used to reason about a particular execution, similar to a trace.

Definition 9 (Path)
Let $G = (V, E, \mathit{entry}, \mathit{exit})$ be a CFG. A path in G is a sequence of vertices $(x_1, ..., x_k) \subseteq V^k$ such that for all j, $1 \leq j < k$, there exists an edge $(x_j, x_{j+1}) \in E$. We adopt the notation $\mathit{Paths}(G)$ for denoting the set of all paths in G. Also, when it is clear from context, we let $(a_1, ..., a_k)$ denote $((a_1, i_1), ..., (a_k, i_k)) \in \mathit{Paths}(G)$.

Next, we use the CFG to create the *Control Dependency Graph* (CDG) and the *Data Dependency Graph* (DDG). These graphs capture information about how instructions rely on each other. The CDG shows which instructions control the execution of other instructions, and is obtained by using a simple post dominance algorithm[8, 15].

The DDG shows which instructions have written the data that is read by other instructions later in the program's execution. This is found with a simple worklist algorithm that computes a reaching definitions analysis[14]. It is important to note that our data dependencies track the use of both registers and flags like in [11].

Before we continue to the final type of dependency graph, we define a couple of useful data dependency related notations.

We first define the functions Reads and Writes which denote the registers and flags that are read and written by instructions:

$$\mathit{Reads} : \texttt{InstrA} \rightarrow 2^{\mathit{DataReg} \,\cup\, \mathit{Flag}}$$

is the function that maps instructions to the registers and flags that they read.

$$\mathit{Writes} : \texttt{InstrA} \rightarrow 2^{\mathit{DataReg} \,\cup\, \mathit{Flag}}$$

is the function that maps instructions to the registers and flags that they write to.

Remark, that fault instructions do not read any register, but writes a single register (their target) and this is reflected in the definition of Reads and Writes.

We use these functions to provide a formal definition of data dependencies.

Definition 10 (Data Dependency)
Given a CFG $G = (V, E, \mathit{entry}, \mathit{exit})$ and two vertices $u, v \in V$, we say that v is *data dependant* on u, written $u \xrightarrow{d} v$, if and only if the following holds:

1. $\mathit{Writes}(u) \cap \mathit{Reads}(v) \neq \emptyset$
2. there exists a register or flag, $s \in \mathit{Reads}(v)$, and a path $(u, x_1, ..., x_k, v)$ such that for all i, where $0 < i \leq k$, we have that $s \notin \mathit{Writes}(x_i)$.

Note that the definition of data dependencies extends to fault instructions as well. This implies that any instruction that reads the value of a register directly written by a fault instruction is data dependant on the fault instruction.

The CDG and DDG are used to build a composite dependency graph called the *Program Dependency Graph* (PDG), which is simply the combination of the CDG and DDG. The purpose of the PDG is to create the backwards slice of the program. An example of a PDG can be found in Example 8.

Example 8 (Dependency Graphs)
Consider the alarm controller program in Listing 7.

```
1  LDR r0, 0xFEED   ; Read sensor.
2  MOV r1, 0        ; Min. error bound.
3  MOV r2, 10000    ; Max. error bound.
4  MOV r3, 2000     ; Dangerous temp.
5  CMP r0, r1       ; Check min.
6  BLT noalarm
7  CMP r0, r2       ; Check max.
8  BGT noalarm
9  CMP r0, r3       ; Check danger.
10 BLT noalarm
   alarm:
   ...
11 B exit
   noalarm:
12 ASSERT r0@1 < 2000 ∨ r0@1 > 10000
   ...
   exit:
```

Listing 7: Alarm controller used to demonstrate dependency graphs.

The PDG for this program can be seen in Figure 5. As we mentioned earlier, the PDG is a combination of the CDG and the DDG. In this figure, the edges from the CDG are dashed lines, while the edges from the DDG are solid lines. The main purpose of this graph is to compute the backwards program slice from the security assertion. This is explained below.

A backwards slice of a program is the subset of the program's instructions that influence a target instruction in some way:

Definition 11 (Backwards Slice)
Let $G = (V, E, entry, exit)$ be a CFG. The backwards slice of G to $t \in V$ is a subset of vertices $S_t(G) \subseteq V$ where $t \in S_t(G)$

We usually utilise the security assertion of a program as the target instruction, which enables us to find all instructions that control the flow towards the critical code of the program. By using a backwards slice we do not have to analyse the entire target program, since some instructions may not affect the program's security. This is useful when we want to select which faults should be tested, since faults outside the slice cannot affect the control flow to the critical program point. Furthermore, the slice is useful during symbolic execution, since it allows us to skip all the instructions that are not in the slice.

To find the backwards slice, we do an exhaustive backwards search in the PDG from the security assertion. This finds all the instructions of the backwards slice. It is important to note that our backwards slice overapproximates which instructions influence the target. This means

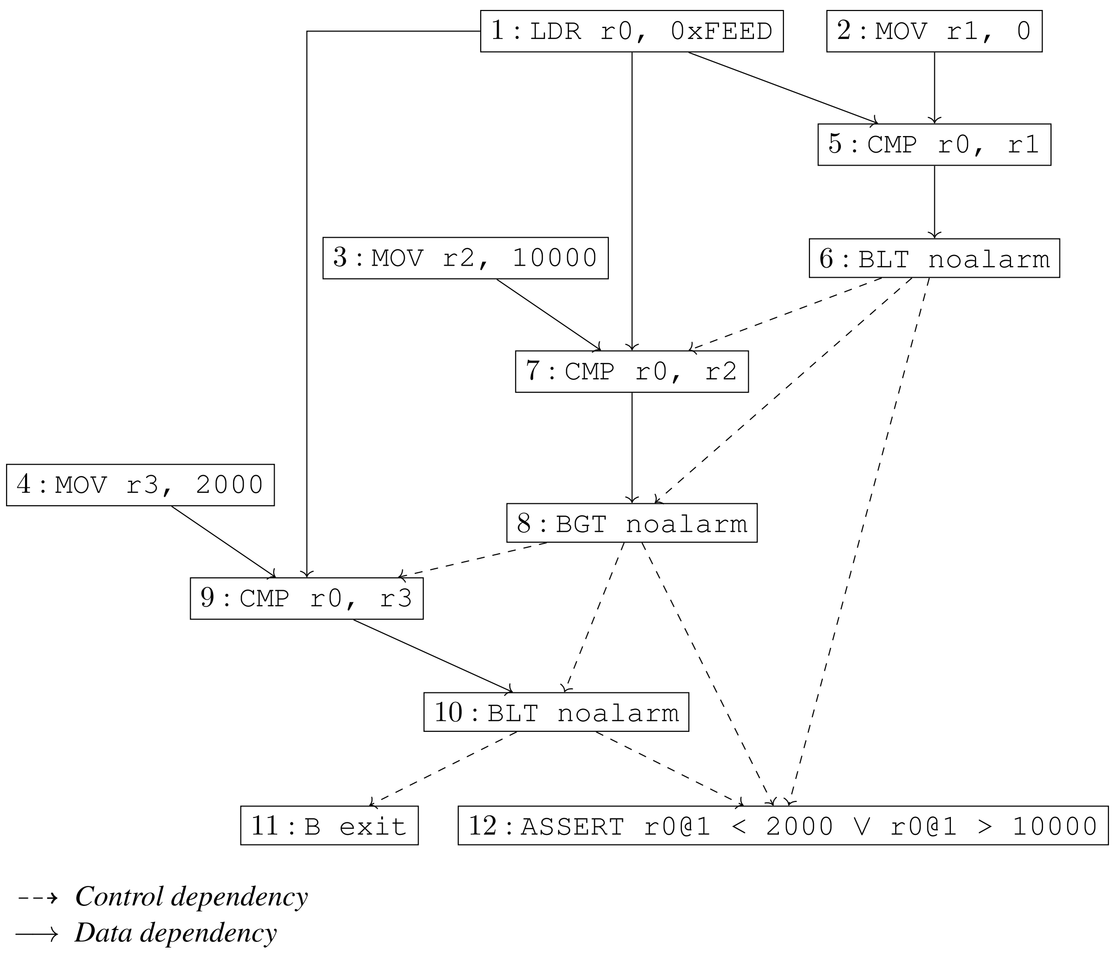


Figure 5: PDG of Listing 7.

that potential faults cannot change control flow to leave the backwards slice and re-enter it to reach the target. If we compute the backwards slice for the program in Example 8, we can see that all instructions, expect instruction 11, are in the slice. The reason for this is that the `alarm` code only contains one instruction. In a real world applications there would be more instructions in the `alarm` code and the backwards slice would be more beneficial since it removes more instructions from the analysis.

8 Vulnerability Analysis

We now present how to analyse each fault for possible vulnerabilities. As mentioned in Section 6 we focus the analysis on a subset of all faults in order to speed up the analysis, since many faults are irrelevant for a program's execution (for instance a fault targeting an unused register or a fault occurring immediately before a register is overridden by a `MOV` instruction). Additionally, since we are interested in the possible effects that faults can cause, and not each individual fault's effect, we ensure that only distinct faults are considered. That is, we ensure that only one representative fault from each fault equivalent class is selected for further analysis.

In summary, the vulnerability analysis consists of the following steps:

1. Find initial fault candidates.
2. Check each initial fault for vulnerability by collecting and solving the path conditions from by symbolic execution.
3. Check vulnerable faults for fault equivalence and select one representative from each equivalence class.

The result is the set of non-equivalent faults that can cause a vulnerable trace in the program. This set of faults are the basis for the quantitative analysis.

Step 1: Finding initial faults

The analysis starts by searching for registers used in the backwards program slice to the target instruction. The goal is to remove trivially irrelevant faults, i.e. faults that target unused registers as well as some faults that are trivially equivalent. Although simple, this step eliminates a large amount of irrelevant faults that would only cause unnecessary performance overhead.

The initial faults are the faults occurring immediately before their target register is read. That is, there exist no intermediate fault instructions that target the same register on the same bit position between execution of the initial fault and the next instruction that reads the target register. Formally, the set of initial faults w.r.t. a slice S and a CFG G is specified as follows:

$$\begin{aligned} F(S,G) = \{f_a \mid\ & (a, \texttt{FAULT r, i}) \in S \land (c, \mathit{instr}) \in S \land \\ & (a, \texttt{FAULT r, i}) \xrightarrow{d} (c, \mathit{instr}) \land \\ & \forall \pi = ((a, \texttt{FAULT r, i}), ..., (c, \mathit{instr})) \in \mathit{Paths}(G) : \\ & \quad \forall x \in \pi : x = (b, \texttt{FAULT r, i})\ \mathit{implies}\ a = b\} \end{aligned}$$

Example 9
Consider the alarm controller program in Listing 8. We have only included a small subset of all possible faults, since it is sufficient to illustrate how initial candidates are found.

Recall that f_a denotes that the fault at address a is enabled and every other fault is disabled. We now consider each fault:

- $f_5 \notin F(S,G)$ since no instruction is data dependant on the fault in instruction 5.
- $f_6 \in F(S,G)$ since $6 \xrightarrow{d} 8$ and there exist no other instance of this fault on the path from instruction 6 to instruction 8.
- $f_7 \in F(S,G)$ since $7 \xrightarrow{d} 8$ and no other instance of the fault instruction exist on a path from instruction 7 to instruction 8.

- $f_{10} \notin F(S,G)$ since no instructions are data dependant on the fault in instruction 10 (`r1` is not live after instruction 8).
- $f_{11} \notin F(S,G)$ since there exists another instance of the same fault (instruction 14) along the path to instruction 16.
- $f_{14} \in F(G,S)$ since $14 \xrightarrow{d} 16$ and no other instances of the same fault exist on the path to instruction 16.
- $f_{15} \in F(G,S)$ for the same reason as f_{14}.

In summary, we obtain the initial fault candidates: $F(S,G) = \{f_6, f_7, f_{14}, f_{15}\}$.

```
1   LDR r0, 0xFEED    ; Read sensor.
2   MOV r1, 0         ; Min. error bound.
3   MOV r2, 10000     ; Max. error bound.
4   MOV r3, 2000      ; Dangerous temp.
5   FAULT r4, 2
6   FAULT r0, 4
7   FAULT r1, 1
8   CMP r0, r1        ; Check min.
9   BLT noalarm
10  FAULT r1, 7
11  FAULT r3, 13
12  CMP r0, r2        ; Check max.
13  BGT noalarm
14  FAULT r3, 13
15  FAULT r3, 14
16  CMP r0, r3        ; Check danger.
17  BLT noalarm
    alarm:
    ...
18  B exit
    noalarm:
19  ASSERT r0@1 < 2000 ∨ r0@1 > 10000
    ...
    exit:
```

Listing 8: Alarm controller with faults.

Some of the initial faults may still be equivalent, such as the ones in the small example in Figure 6. This can happen because unoptimized assembly code often have redundant instructions that just move data between registers. These trivially equivalent faults will be handled by Step 3, where the analysis eliminates any remaining equivalent faults. Alternatively, they can also be handled by a separate program analysis that removes the redundant instructions from the program.

It is important that the procedure for finding the initial fault candidates is sound in the sense that every fault equivalence class with respect to the slice is represented. Otherwise, we are missing entire classes of effects that could cause serious unintended behaviour. While we provide no formal proof of soundness, we claim that this is true for F.

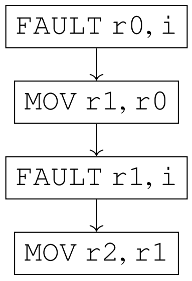


Figure 6: Equivalent faults not eliminated by Step 1.

Claim 1
Let $G = (V, E, \mathit{entry}, \mathit{exit})$ be a CFG and $S_u(G)$ be a slice of G to $u \in V$, then $F(G, S)$ is a sound approximation of distinct faults in w.r.t. to S.

Step 2: Symbolic Execution

After the initial fault candidates are found by Step 1, we test the vulnerability of each fault by collecting the path conditions with symbolic execution and checking whether the security assertion can be broken. If the assertion can be broken, then the fault caused an unintended control flow to reach critical code. Any fault that can do this is of interest to the analysis, and the rest of the fault candidates are discarded. The pseudocode of this procedure is shown in Algorithm 1.

Algorithm 1 Finding faults that break the security assertion.

```
Input: A starting configuration, $\langle P, pc, R, F, C\rangle$. The initial faults, $F$. The security assertion address, $t \in Addr$.
Output: A set of vulnerable faults.
1: function VulnerableFaults($\langle P, pc, R, F, C\rangle,\ F,\ t$)
2:     $V \leftarrow \emptyset$
3:     for all $f \in F$ do
4:         if $f \vdash \langle P, pc, R, F, C\rangle \overset{s}{\Longrightarrow}{}^{*} \langle P, t, R', F', C'\rangle$ then
5:             if CheckSat($C'(t)$) $=$ true then
6:                 $V \leftarrow V \cup \{f\}$
7:     return $V$
```

The call to the `CheckSat` is an abstraction of the invocation of the chosen SMT solver - in our case, the Z3 SMT solver.

Example 10 (Symbolic Execution)
Consider again the alarm controller program in Listing 8. Recall that we found the initial fault candidates in Example 9 to be $F(S, G) = \{f_6, f_7, f_{14}, f_{15}\}$. We now use symbolic execution to check the vulnerability of each fault.

Like earlier we represent the symbolic execution of a program as a tree. The tree, which can be seen in Figure 7, is similar to the tree in Example 4 but uses other addresses since Listing 8 contains more fault instructions.

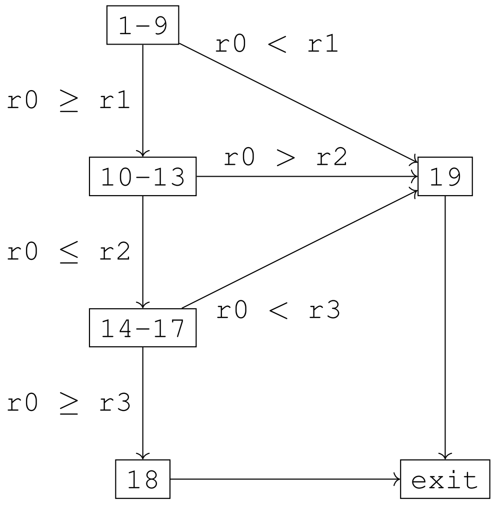


Figure 7: Symbolic execution of Listing 8.

Symbol	**Given By**	**Value**
s_0	*Input*	*Anything*
s_1	$s_1 \equiv_1^4 s_0$	*Anything*
s_2	$s_2 \equiv_1^1 0$	2
s_3	$s_3 \equiv_1^{13} 2000$	10192
s_4	$s_4 \equiv_1^{14} 2000$	18384

Table 1: Values of the symbols generated by symbolic execution of Listing 8.

Several symbols will be generated by the symbolic execution of the program. These symbols are summarised in Table 1. We can see that the symbol s_0 represents the initial input to the program, while the symbols s_1, s_2, s_3, and s_4 are generated by the fault instructions.

We now symbolically execute the program for each initial fault to build the path conditions collected on the path to the security assertion. We then solve the collected path conditions to check whether they are satisfied. If the path conditions are satisfied, we mark the fault as vulnerable.

We start with f_6. The path conditions collected to the security assertion are:

$$s_1 \equiv_1^4 s_0 \wedge s_1 \geq 0 \wedge s_1 \leq 10000 \wedge s_1 < 2000$$

This formula is satisfied when s_1 is in the interval [0 .. 1999]. We now add the negated security assertion (like in Example 4) and check whether the conditions can still be satisfied:

$$\begin{aligned} &s_1 \equiv_1^4 s_0 \wedge s_1 \geq 0 \wedge s_1 \leq 10000 \wedge s_1 < 2000 \wedge \neg(s_0 < 2000 \vee s_0 > 10000) = \\ &s_1 \equiv_1^4 s_0 \wedge s_1 \geq 0 \wedge s_1 \leq 10000 \wedge s_1 < 2000 \wedge s_0 \geq 2000 \wedge s_0 \leq 10000 \end{aligned}$$

where s_0 is the value of `r0@1`. This formula is satisfied if s_0 (the input) is in the interval [2000 .. 10000], s_1 (the corrupted value) is in the interval [0 .. 1999], and $s_1 \equiv_1^4 s_0$. We see that it is possible to satisfy the conditions with the assignment $s_0 = 2000$ and $s_1 = 1984$. Since the path conditions are satisfiable, f_6 can break the security of the program and is returned by Algorithm 1.

For f_7 we collect the following path conditions:

$$s_4 \equiv_1^1 0 \land s_0 \geq s_2 \land s_0 \leq 10000 \land s_0 < 2000 \land \neg(s_0 < 2000 \lor s_0 > 10000) =$$
$$s_1 \equiv_1^1 0 \land s_0 \geq s_2 \land s_1 \leq 10000 \land s_0 < 2000 \land s_0 \geq 2000 \land s_0 \leq 10000$$

Observe that $s0 < 2000 \ \land \ s0 \geq 2000$ is a contradiction, hence the path conditions are not satisfiable and f_7 is not vulnerable.

The fault functions f_{14} and f_{15} can also break the security assertion of the program. This can be seen if we look at their path conditions:

$$C_{f_{14}} = s_0 \geq 0 \land s_0 \leq 10000 \land s_3 \equiv_1^{13} 2000 \land s_0 < s_3 \land s_0 \geq 2000 \land s_0 \leq 10000$$
$$C_{f_{15}} = s_0 \geq 0 \land s_0 \leq 10000 \land s_4 \equiv_1^{14} 2000 \land s_0 < s_4 \land s_0 \geq 2000 \land s_0 \leq 10000$$

In the path conditions for both of these fault functions, s_0 has to be in the interval $[2000 .. 10000]$ which is possible since it is fully symbolic. Furthermore, s_0 should be less than the value of the symbols generated by the faults (s_3 and s_4). This is always the case since both s_3 and s_4 are greater than 10000 (see Table 1) while $s_0 \leq 10000$. Because of this, both f_{14} and f_{15} can break the security and they are returned by Algorithm 1.

In summary the fault functions f_6, f_{14}, and f_{15} can break the security of the program and are returned by Algorithm 1. Fault function f_7 cannot break the security so it is not returned by Algorithm 1.

Step 3: Equivalence Testing

The third and final step of the vulnerability analysis is to eliminate the remaining equivalent faults according to Definition 7. This is done to further reduce the number of faults that are considered during quantitative analysis.

Recall from Definition 7 that in order for two faults to be equivalent, the conditional flags must be identical for every intermediate configuration in traces from the same starting configuration. Since the vulnerability analysis considers only traces in the slice to the security assertion, it is sufficient to check whether the possible contents of the flags are identical for instructions in the slice. For testing fault equivalence, we leverage the path conditions already collected by the symbolic execution of each vulnerable fault. Since the contents of the flags are collected by symbolic execution, we can construct a formula expressing that the conditional flags are identical and check whether it holds for every program point. More specifically, we test the equivalence of two faults by constructing *combined path conditions* for each program point. The combined path condition consists of the condition that the expressions in the flag environments must be identical together with the base path conditions generated by the symbolic execution, i.e. the conditions that in general must hold to reach a program point. We prove or disprove that the combined path conditions hold *for all* possible runs, by checking satisfiability of its negation with an SMT solver. If the negation of the combined path conditions are satisfiable then there exist some input, i.e. some execution, where the flags are *not* identical, hence we get that the faults are *not* equivalent.

The main function for equivalence testing is shown in Algorithm 2. This function iteratively

Algorithm 2 Computing distinct faults.

```
    Input: Program $P \in \mathit{Program}$. Set of faults $F$. Target $t \in \mathit{Addr}$.
    Output: A set of distinct faults.
 1: function DISTINCTFAULTS($P,\ F,\ t$)
 2:     $f \in F$
 3:     $D \leftarrow \{f\}$
 4:     for all $f_a \in F \setminus \{f\}$ do
 5:         if $\forall f_b \in D : \textsc{AreEquivalent}(P, f_a, f_b, t) = \mathit{false}$ then
 6:             $D \leftarrow D \cup \{f_a\}$
 7:     return $F$
```

checks whether the initial faults are equivalent to an already tested set of distinct faults. DISTINCTFAULTS calls the function shown in Algorithm 3 to determine if two faults are equivalent.

Algorithm 3 Testing equivalence of two faults.

```
    Input: Program $P \in \mathit{Program}$. Faults $f_a, f_b \in \mathit{Addr} \to \{\mathit{true}, \mathit{false}\}$. Target $t \in \mathit{Addr}$.
    Output: $\mathit{true}$ if equivalent; otherwise, $\mathit{false}$.
 1: function AREEQUIVALENT($P, f_a, f_b, t$)
 2:     for all $T_1 \in f_a \vdash \langle P, pc, R_0, M_0, F_0, C_0 \rangle \overset{s}{\Longrightarrow}{}^* \langle P, t, R_k, M_k, F_k, C_k \rangle$ do
 3:         for all $T_2 \in f_b \vdash \langle P, pc, R_0, M_0, F_0, C_0 \rangle \overset{s}{\Longrightarrow}{}^* \langle P, t, R'_k, M'_k, F'_k, C'_k \rangle$ do
 4:             if $\textsc{Path}(T_1) = \textsc{Path}(T_2)$ then
 5:                 for all $i \in \textsc{Path}(T_1)$ do
 6:                     $\mathit{combined} \leftarrow C_k(i) \wedge C'_k(i) \wedge \neg(F_k(i) = F'_k(i))$
 7:                     if $\textsc{CheckSat}(\mathit{combined}) = \mathit{true}$ then
 8:                         return $\mathit{false}$
        return $\mathit{true}$
```

Algorithm 3 tests each trace with the same path in the respective symbolic execution under f_a and f_b. Note that the symbolic execution is not completed again, but the path conditions are simply taken from the previous symbolic execution in Step 2 of the vulnerability analysis.

For each instruction along the traces, the combined path conditions are constructed (Line 6) by asserting both base path conditions and that the flags are *not* equal. Recall that flag environments are functions, therefore the equality assertion on $F_k(i), F^i_k(i)$ is done point-wise.

If the combined path conditions are satisfiable, it is possible for the flags to differ which implies $f_a \not\equiv^P_f f_b$ and the call to AREEQUIVALENT therefore immediately returns false. If none of the combined path conditions are satisfiable, we have that the faults are equivalent.

In the above algorithms for equivalence testing, we check all program points in the path to the target security assertion. Implementation wise, it is sufficient to check the combined path conditions after every flag setting instruction along the trace, since other instructions will keep the flags unchanged.

Example 11 (Equivalence Testing)
Recall the alarm controller program from Listing 8. Suppose that we want to determine whether any of the faults from Step 2 (f_6, f_{14}, and f_{15}) are equivalent.

There are three flag setting instructions (8, 12, 16), so it is sufficient to check the possibility of the flags differing *after* every of those program points. First, we assert the path conditions under each fault function for the given program point. Secondly, we assert that the flags under each fault function must be identical.

Address	**Fault**	**Flags**
8	f_6	$\mathit{flagsSe}_{\texttt{CMP}}(s_1,\ 0,\ F)$
	f_{14}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F)$
	f_{15}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F)$
12	f_6	$\mathit{flagsSe}_{\texttt{CMP}}(s_1,\ 10000,\ F)$
	f_{14}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 10000,\ F)$
	f_{15}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 10000,\ F)$
16	f_6	$\mathit{flagsSe}_{\texttt{CMP}}(s_1,\ 2000,\ F)$
	f_{14}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ s_3,\ F)$
	f_{15}	$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ s_4,\ F)$

Table 2: Flag values after each `CMP` instruction in Listing 8.

We start by checking whether $f_6 \equiv_f^P f_{14}$. We check the path and flag conditions under each fault function after the comparison at address 8. The flag condition can be found in Table 2 and the path conditions are the following (after contracting the multiple true conjunctions):

$$\begin{aligned} \mathit{Path}_{f_6} &= \mathit{true} \land s_1 \equiv_1^4 s_0 \\ \mathit{Path}_{f_{14}} &= \mathit{true} \end{aligned}$$

Combining the path conditions (except the trivial true) and the negated flag conditions, we get the following:

$$\mathit{Path}_{f_6} \land \mathit{Path}_{f_{14}} \land \mathit{flagsSe}_{\texttt{CMP}}(s_1,\ 0,\ F) \neq \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F)$$

This can be satisfied if, for instance, $s_0 = 0$ as this would make the zero flag false in $\mathit{flagsSe}_{\texttt{CMP}}(s_1,\ 0,\ F)$ but true in $\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F)$. Because the flags can be different in one program point, we do not need to test the rest since we already know that $f_6 \not\equiv_f^P f_{14}$.

Lets skip ahead and check whether $f_{14} \equiv_f^P f_{15}$. Again we start by checking the path and flag conditions after instruction 8. The flag condition is found in Table 2. Since we have not visited any conditional or fault instructions, the path conditions under both faults will be the trivial true conjunctions, so we ignore these. The combined conditions will be:

$$\mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F) \neq \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 0,\ F)$$

which is clearly a contradiction. Therefore we need to continue to the next comparison instruction at address 12.

The path conditions at instruction 12 are the same under both of the fault functions: $s_0 \geq 0$ (excluding the trivial true conjunctions). This gives us the combined conditions:

$$s_0 \geq 0 \land s_0 \geq 0 \land \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 10000,\ F) \neq \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ 10000,\ F)$$

Again, it is immediately clear that the combined path conditions are unsatisfiable.

When we reach the third comparison instruction at address 16, the paths under the fault functions will be (again excluding the true conjunctions):

$$\begin{aligned} \mathit{Path}_{f_{14}} &= s_0 \geq 0 \land s_0 \leq 10000 \land s_3 \equiv_1^{13} 2000 \\ \mathit{Path}_{f_{15}} &= s_0 \geq 0 \land s_0 \leq 10000 \land s_4 \equiv_1^{14} 2000 \end{aligned}$$

The combined path and flag conditions after the instruction are:

$$\mathit{Path}_{f_{14}} \land \mathit{Path}_{f_{15}} \land \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ s_3,\ F) \neq \mathit{flagsSe}_{\texttt{CMP}}(s_0,\ s_4,\ F)$$

While it is easy to see that the two path conditions are satisfiable, it is not immediately clear whether the flag condition is satisfiable or not. Observe that when we reach instruction 16, we know that $s_0 \leq 10000$ by the path conditions. From Table 1 we also know that $s_3 = 10192$ and $s_4 = 18384$. By inputting these values into the calls to the $\mathit{flagsSe}$ function, we can compute the possible flag values under each fault function:

$$f_{14} : \begin{cases} f_N &= \mathit{true} \\ f_Z &= \mathit{false} \\ f_C &= \mathit{false} \\ f_V &= \mathit{false} \end{cases} \qquad f_{15} : \begin{cases} f_N &= \mathit{true} \\ f_Z &= \mathit{false} \\ f_C &= \mathit{false} \\ f_V &= \mathit{false} \end{cases}$$

We can see that the possible values of the flags are the same under each faults function. This means that the condition where the flags should be different is unsatisfiable and therefore the combined conditions are also unsatisfiable. Since this is the last flag setting instruction we now know that f_{14} and f_{15} are equivalent.

In summary, the distinct faults returned by Algorithm 2 are f_6 and either f_{14} or f_{15}.

We have given a symbolic execution based procedure for testing fault equivalence. It is a natural choice for our implementation, as we have already collected the necessary analysis data needed from the previous symbolic execution step.

An alternative procedure for determining fault equivalence is to leverage techniques from model checking. We propose to encode two programs as a UPPAAL model - each with their respective fault encoded as a fault instruction location. With the two models at hand, we can then verify whether it is possible to get to the "same" location in each program, while the content of the code-behind flags are not the same. To increase performance, it may be advantageous to make sure that the transitions of the two programs synchronize, such that locations representing different program points are not checked against each other.

9 Quantitative Analysis with UPPAAL SMC

In this section we present how to quantify the risk of each of the vulnerable faults found by the vulnerability analysis. Accurate estimation of the risk for each fault is a desirable metric for a multitude of fault tolerance applications.

We leverage *statistical model checking* capabilities of the model checking tool UPPAAL[5] to perform quantitative analysis of the programs. Using SMC provides us with better control over the experiments than physical fault injection experiments. Furthermore, it scales well and simulation runs can be trivially parallelised and distributed to increase performance.

First, we present how to encode TinyARM control flow graphs and the SEU fault model as UPPAAL models. We then discuss how to query UPPAAL SMC to get the data.

9.1 Modelling TinyARM Configurations

A UPPAAL model is a *timed automaton*. Informally, a timed automaton consists of a set of locations, with one initial location, a set of clocks, clock constraints, as well as a set of actions and edges between locations. In UPPAAL, locations have an associated *invariant* that must be satisfied when the location is part of the current state of the model. Each edge can have associated guard and update statements. The guard must be satisfied before the edge is enabled. If the edge is enabled and traversed, the update statement is executed and the state of the underlying model is changed. The underlying state is coded in a C-like language and can be constructed to closely represent the actual system being modelled. The state of the TinyARM abstract machine is modelled in a straightforward manner in the underlying UPPAAL model state. The principal part of the underlying state for the alarm controller program is shown in Listing 9.

```
reg_t r0, r1, r2, r3, r0_4;
bool fn, fz, fc, fv;
bool hit = false;
int16_t faultReg;
int16_t faultAddr;
int16_t faultBit;
```

Listing 9: Example of underlying state of a TinyARM program.

The register environment is represented by 16-bit signed integer variables. Since we already have knowledge about the structure of the program from previous analysis steps, it is sufficient to model the registers that are live in the program. Unfortunately, UPPAAL does not directly support 16-bit integers, so we model them using bounded 32-bit integers and handle overflow and carry manually (in calls to a `fixOverflow` function).

Any registers referenced by the security assertion are also stored in a special variable, so that we later can reference them in the model of the security assertion. In Listing 9 the special register for the alarm controller program is `r0_4`. The conditional flags are simply represented by four boolean variables.

During quantitative analysis, the `hit` variable is used to signify if a fault occurs or not. The `faultReg`, `faultAddress` and `faultBit` variables store register, address, and bit position targeted by the enabled fault, respectively. We store these values to get data about the frequency of each fault through the simulation.

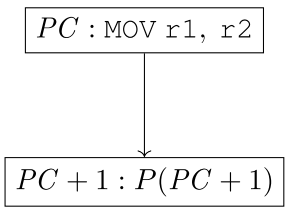
9.2 Modelling TinyARM Semantics

Recall that a vertex in a CFG represents a single instruction in the program and an edge represents the control flow between instructions. Similarly, we can represent a program as a UPPAAL model by constructing a location for each vertex and an edge for each edge in the CFG. To closely model the TinyARM semantics, the premises of the semantic rules are set as the guards, and the changes to registers and flags are modelled by the update statement on the edges. Essentially, this means that when a location transitions to another, the effects of the instruction semantics are applied to the underlying state.

A CFG also contains two special vertices: the entry vertex and the exit vertex, which represent the entry and exit points of the program. The entry and exit vertices of the CFG are also added to the UPPAAL model to mark the models' entry and exit points, but they have no other effect on the model.

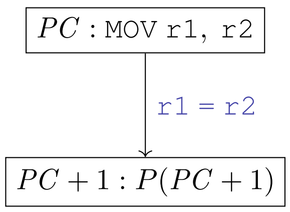
We now show how to encode some of the TinyARM instructions in UPPAAL. We start with the `MOV` instruction, since this is a simple instruction that just updates a single register.

Consider the CFG in Figure 8a and the UPPAAL model in Figure 8b. Most transitions in the UPPAAL models also updates a program counter variable, but to keep the following figures simple, the program counter updates are not shown.



$$\begin{tikzcd} \boxed{PC: \texttt{MOV r1, r2}} \arrow[d] \\ \boxed{PC+1: P(PC+1)} \end{tikzcd}$$

(a) `MOV` instruction in a CFG.



$$\begin{tikzcd} \boxed{PC: \texttt{MOV r1, r2}} \arrow[d, "\texttt{r1 = r2}"] \\ \boxed{PC+1: P(PC+1)} \end{tikzcd}$$

(b) `MOV` instruction in a UPPAAL model.

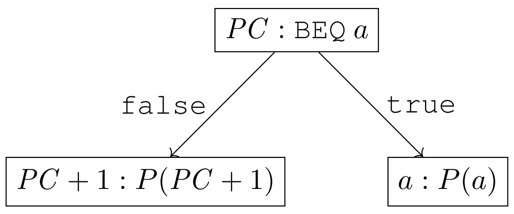
Figure 8: CFG conversion rule for `MOV`.

We can see that the UPPAAL model looks very similar to the CFG. The only difference is the update statement on the edge which updates the value of `r1` to be the same as the value of `r2`. This is marked with a blue colour. This updates the variables used by the UPPAAL model according to the `MOV` semantics. Conditional flags are set in the same way by the flag setting instructions.

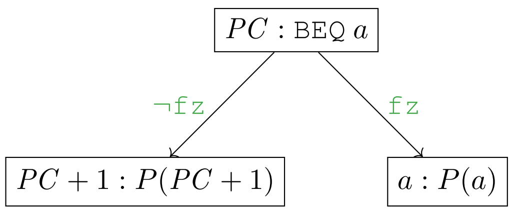
In order to model conditional execution of instructions in UPPAAL, we use guards on the edges. An edge with a guard is only enabled if the guard evaluates to true. An example of this can be seen in the conditional branch in Figure 9a and Figure 9b.

Here we can see how guards are used to control the flow in the model. The guards are marked with a green colour. Instructions with the condition code `EQ` (equals) are executed if the zero flag (`fz`) is set. We model this in UPPAAL by using the zero flag as a guard on the edges. This means that the model can only execute the branch (transition to the instruction at a) if the zero flag is set. Otherwise, the model transitions to the next instruction.

Other conditional instructions are encoded in a similar manner, except they will visit the subsequent instruction with both their outgoing edges. An example of this can be seen in Figure 10.



(a) `BEQ` instruction in a CFG.



(b) `BEQ` instruction in a UPPAAL model.

Figure 9: CFG conversion rule for conditional branch.

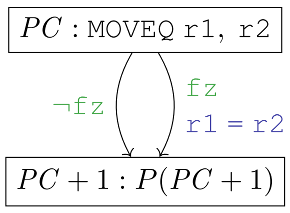


Figure 10: `MOVEQ` instruction in a UPPAAL model.

We can see that both edges from the `MOVEQ` instruction transition to the same instruction, but with different guards. Furthermore, the guard that models the case where `EQ` evaluates to true (`fz`) will have the assignment to `r1`. The reason for this is that the assignment should only happen if the instruction executes (i.e. if `EQ` is true).

We use the `LDR` instruction of TinyARM as an input mechanism for a program. To model this in UPPAAL we nondeterministically select an input value through a select statement. Note that this implies that input values are uniformly distributed. However, for user-specific applications, domain knowledge of the distribution of inputs may refine the quantification results substantially.

If two instructions load from the same address, we cache the value from the first instruction such that we can use it again in the second instruction. We can see how this works in Figure 11.

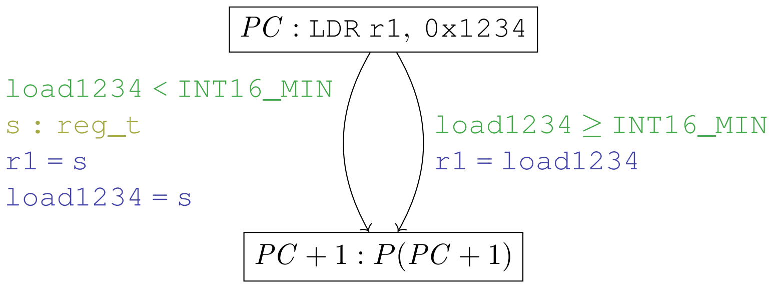
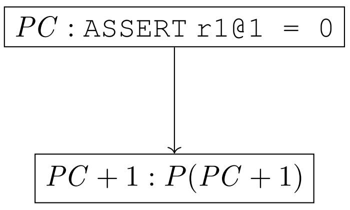


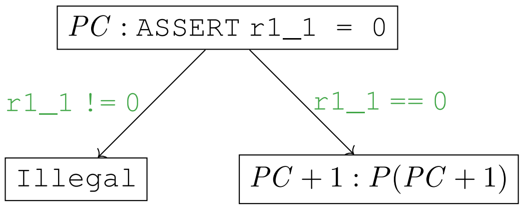
Figure 11: `LDR` instruction in UPPAAL model.

Every address being loaded from has an associated caching variable (`load1234` in Figure 11). The initial values of these cache variables are lower than the lowest 16-bit value. This allows us to check whether they are in use by using a guard that compares their current value against the lowest 16-bit value (`INT16_MIN`). If the cache variable has not been used yet, we know that this is the first time loading from the given address. We do this by nondeterministically selecting a 16-bit value and assigning this to the given register as well as the cache variable. The next time we load from the same address, we simply use the cached value instead of selecting a new one. In this way, we make sure that the memory is consistent throughout each simulation run. Note that this is based on the assumption that memory is reliably protected through some hardware mechanism, such that faults do not occur in memory. In order to handle fault models with faults in memory, it will be necessary to extend the analysis to handle the otherwise fully symbolic memory.

The last instruction that we show the conversion for is the `ASSERT` instruction which works a little differently in the UPPAAL model. When we meet an `ASSERT` instruction we want to check whether the flow is legal or illegal based on the condition in the security assertion. To do this, we add two locations and place the `ASSERT` condition as a guard on the incoming edges. This can be seen in Figure 12a and Figure 12b.



(a) `ASSERT` instruction in a CFG.



(b) `ASSERT` instruction in a UPPAAL model.

Figure 12: CFG conversion rule for `ASSERT`.

We can see that there are some differences between the two figures. The CFG only has an outgoing edge from the `ASSERT` instruction to the next instruction in the program, while the UPPAAL model has two outgoing edges. The UPPAAL model has an edge to the next instruction and an edge to a location named `Illegal`. By using this model, we can check whether the flow was illegal, as this would cause the model to deadlock in the `Illegal` location. While other work [9, 7] take the approach continuously of observing the effects of faults on the machine state, we can avoid such complex observer components since we focus on fault tolerance with respect to a specific program point, and not the whole program.

The complete UPPAAL model for the alarm controller program is shown in Appendix D.

9.3 Modelling Faults in UPPAAL

Modelling the core language semantics of TinyARM programs is only one part of the quantitative analysis. We must also model how faults occur during program execution, such that simulation represents realistic fault risks.

We use two different SEU models which we call the *Implicit SEU Model* and the *Explicit SEU Model*.

The implicit SEU model represents that a single fault on any register occurs randomly with uniform distribution throughout the execution of the program. This closely represents how SEUs happen in real world applications, and we use it to establish a baseline probability distribution of faults for our experiments in Section 10. The implicit model represents the base-line risk assessment of faults and we use it for comparison with risk assessment of experiments with the explicit SEU model.

The explicit SEU model encodes faults explicitly in the program structure as fault instructions, as we have presented in previous sections. Exactly one of the encoded faults are nondeterministically chosen to be enabled during the execution of the program. The principal difference between the implicit and explicit SEU model is that the explicit model encodes only the set of non-equivalent faults that were determined to be vulnerable by the vulnerability analysis step, while the implicit model allows faults to occur in registers that are not live or in bits that are not vulnerable.

Implicit SEU Model

The implicit SEU model consists of three UPPAAL components: two copies of the program under analysis, $P1$ and $P2$, as well as a separate SEU model component that performs the effects of a fault occurring during execution. The program components do *not* contain any explicit fault instruction. The reason for this is that the occurrence of faults is implied by the interleaved execution of the separate SEU component and the program model. An example of the models used by the implicit SEU model can be seen in Example 12.

Example 12
We can see how the $P1$ and $P2$ models work by looking at Figure 13 and Figure 14. The models synchronize on `done`, which ensures that $P2$ is not executed before $P1$ has finished its execution.

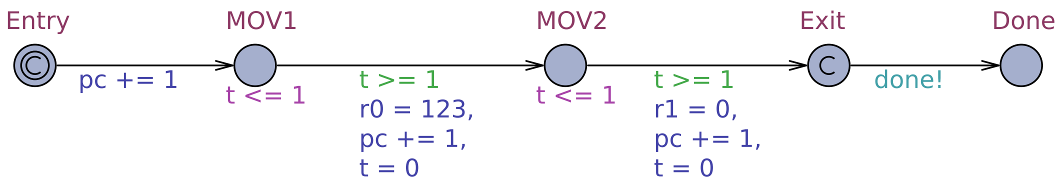


Figure 13: Example of $P1$.

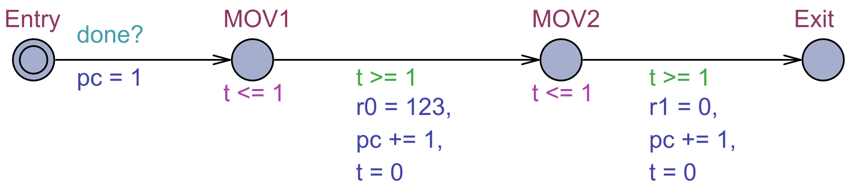


Figure 14: Example of $P2$.

The SEU model for this simple program can be seen in Figure 15. This model also synchronize on `done` to ensure that it does not start before $P1$ finishes. This model also uses edges with probabilistic weights. These weights are coloured orange.

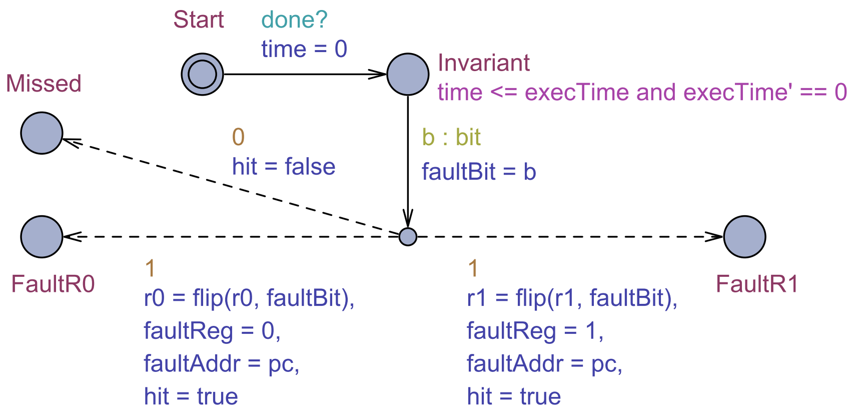


Figure 15: Example of implicit SEU model.

In general, the probability of a fault occurring must be uniformly distributed over the whole execution of the program. We ensure this by performing the following steps for any simulation run:

1. Input is nondeterministically chosen.
2. $P1$ is run with the input and the execution time is saved in a clock variable $execTime$. $P1$ synchronizes on `done` to signal completion.
3. $P2$ is run on the same input as $P1$, interleaved with the SEU component.
4. During the execution of $P2$, the time for the fault to occur is uniformly chosen from $[0, execTime]$. This is done by the invariant in the `Invariant` location of the SEU model.

Note that the execution time for the two programs should be the same for the same input, such that we can ensure that a fault occurs before $P2$ terminates. Otherwise, if $P1$ executes fast and $P2$ slow, we may get a biased distribution of faults towards instructions in the beginning of the program. To ensure identical execution time for $P1$ and $P2$, we enforce that each location of the program delays exactly 1. This is facilitated by a clock variable t, and adding the invariant $t \leq 1$ to every location and every outgoing edge are updated with the guard $t \geq 1$ as well as an update statement resetting t to 0. Intuitively, this means that each instruction takes 1 time instance to execute. Note that in real-world systems, instruction latency depends on various factors such as the type of arithmetic, loads/stores with cache hit/miss from memory, or branching. With this in mind, our scheme for enforcing uniform delays for instructions can be readily adapted to encode the specific latencies for each type of instruction in the processor architecture for the analysed program, ultimately yielding more precise risk quantification of faults.

Remark that since $execTime$ is a clock, we set the clock rate $execTime'$ to 0 in the `Invariant` location in order to stop the clock from delaying any further, essentially making $execTime$ a stopwatch.

When a time has been uniformly chosen, the bit that the fault changes is chosen nondeterministically. The reason that the bit is chosen on the outgoing edge from `Invariant`, and not on each of the probabilistic edges (dashed transitions), is due to the underlying system of UPPAAL SMC models. Here nondeterministic selection on probabilistic transitions generate an edge for each possible value in the domain of bits. Unfortunately, this has the subtle effect of skewing the probabilities in favour of each of the non-missed locations. While this may not have any effect in our case, since the probability of missed is 0, it is a subtle detail that can affect the probability estimations substantially if other weights are used.

A fault is chosen by the SEU model based on the probabilistic weights of the transitions to the fault and miss locations. The effects of the fault are then carried out on the underlying state by a call to the function `flip` shown in Listing 10.

```
reg_t flip(reg_t v, bit b) {
    int32_t res = v ^ (1 << b);
    return fixOverflow(res);
}
```

Listing 10: Flipping a bit in UPPAAL.

The `flip` function computes the new value of the register targeted by a fault by changing the b^{th} bit of v to its complement. This is done by left shifting 1, b times and computing the bitwise exclusive-or of the resulting mask and the value v of the register.

Note that in the SEU model component, the probability weight for `Missed` is 0, since we are analysing the case when a fault occurs with probability 1. These probability weights can be adjusted to model cases where faults are rare, or the probability of faults in some of the registers are greater than others, reflecting any user specific domain knowledge.

Explicit SEU Model

The explicit SEU model follows the fault semantics as well as the fault formalization from Section 3. The UPPAAL model consists of a single program component with all distinct fault instructions, that was determined to be vulnerable by the vulnerability analysis, explicitly encoded as fault instructions. Only one fault is activated per execution and this is facilitated by nondeterministically selecting one fault to activate and making sure only the chosen fault instruction is executed. Originally, we wanted to do this in the beginning of the program (i.e. from the entry location). However, this would lead to cases where faults chosen for testing were never reached, due to input that caused control flow of the program execution to never reach the location of the fault instruction. As a consequence, a large number simulations where in effect invalid, and it would be hard to control the number of usable simulation runs. To overcome this issue, we move the fault selection to the end of the program and execute the program twice. We can see how this works in Figure 16.

Once the program has executed, we reach the `ChooseHit` location in the left of the figure. This location can be reached twice, since we execute the program up to two times. The first time we reach `ChooseHit`, the `hit` variable will be false which tells us that the program has just completed its first run without executing any faults. During this run, the program has recorded all the faults that it visited. This enables us to select a fault that we are sure will

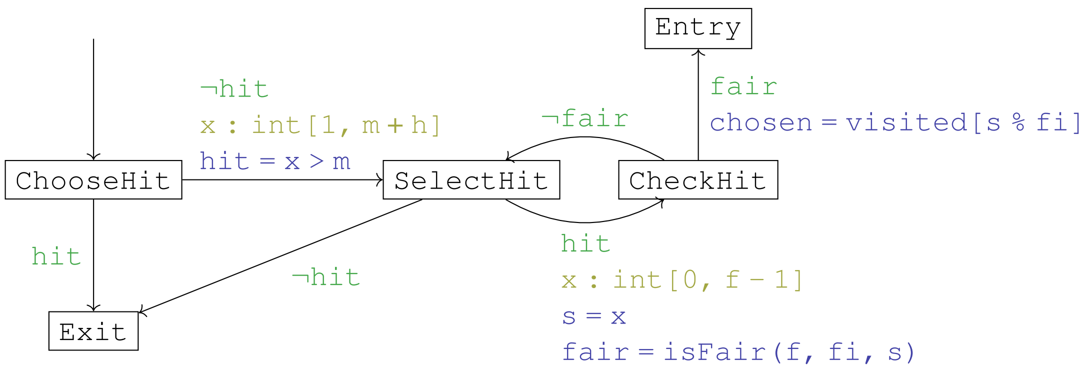


Figure 16: Selecting a fault in the explicit SEU model.

be visited in the second run, as long as the program uses the same input. The second time we reach `ChooseHit`, `hit` will be true and we just transition to `Exit`. The transition from `ChooseHit` to `SelectHit` determines whether we should execute the program a second time with a fault enabled. This transition will select a value from the interval $[1 .. \texttt{m} + \texttt{h}]$, where `m` and `h` are weights for a fault missing or hitting, respectively.

From the `SelectHit` location we have two choices depending on whether a fault should hit or not. If the `hit` variable is false, a fault should not occur in the program, so we just go to `Exit`. If `hit` is true, we need to select one of the visited faults and execute the program a second time with the selected fault enabled. We do this by selecting a random value from the interval $[0 .. \texttt{f} - 1]$, where `f` is the number of fault instructions in the model. The reason for the zero-based interval is that the faults visited in the first execution are stored in a zero-based array. Since we will be selecting a fault based on the selected number and a modulo operation, we first check whether the selected value is *fair* by using the `isFair` function (Listing 11).

```c
bool isFair(int16_t f, int16_t fi, int16_t s) {
    int16_t fullSets = f / fi;
    return s < fi * fullSets;
}
```

Listing 11: Checking whether selected number is fair.

The arguments for the function are: (1) the total number of faults in the model, (2) the number of faults that were visited, and (3) the randomly selected number. We use the `isFair` function to make sure that the randomly selected number has the same chance of choosing any of the visited faults. We can see why this is useful, if we look at the outgoing transitions from the `CheckHit` location.

The `CheckHit` location has two outgoing transitions. One of them loops back to the `SelectHit` in order to select a new fault, if the previous selected fault is not fair. If the selected fault is fair, we can take the transition to the entry location while choosing a fault to enable. In the second run of the program, the selected fault will be executed and we can observe its effect. As we can see, the fault is chosen by taking the modulo of the selected fault number and the number of visited faults in the first run (`fi`). This modulo operation is the reason that we use the `isFair` function, as demonstrated in Example 13.

Example 13
Recall the `isFair` function from Listing 11 and lets imagine a program model with ten fault instructions, i.e. $\texttt{f} = 10$. Lets consider a first run in this program where we visit three fault instructions. This would mean that $\texttt{fi} = 3$ and the `visited` array could be:

```
[1, 2, 3, 0, 0, 0, 0, 0, 0, 0]
```

where `1`, `2`, and `3` are the faults we visited, while the zeros are unused indices in the array.

When we select `s` after `SelectHit`, `s` will be a value in the interval [0 .. 9]. In UPPAAL, select statements need to choose from a range that is known at compile time, and this is the reason that we are not just selecting a value in [0 .. 2]. We can, however, get the selected number to map to the indices of the `visited` array by doing a modulo operation with the number of elements in the `visited` array. This can lead to a bias, however, since some values may be represented more than others.

This is where the `isFair` function is useful. It starts by finding the number of *full sets* that can be mapped to the selected value. In this example the mapping would look like this:

s :	0	1	2	3	4	5	6	7	8	9
fault :	1	2	3	1	2	3	1	2	3	1

where the top row is the selected number, `s`, and the bottom row is the fault that it maps to. We can see that the *full set* of $\{1, 2, 3\}$ appears three times, while the number 1 appears once more than the others. This means that if `s` is in [0 .. 8] we can map it to a fault in an unbiased manner, while $\texttt{s} = 9$ will increase the change of fault 1 happening.

This *fairness* is calculated by the `isFair` function. It starts by calculating the number of full sets by doing an integer division:

$$\texttt{fullSets} = \texttt{f}/\texttt{fi} = 10/3 = 3$$

The value of `fullSets` can then be used to determine whether the selected number is biased or not. Lets see how this works on our example by looking at $\texttt{s} = 4$ and $\texttt{s} = 9$, where we know that 4 is fair while 9 is not. The fairness from the `isFair` function is given by:

```
s < fi * fullSets
```

where we know that $\texttt{fi} = 3$ and $\texttt{fullSets} = 3$.

By substituting `s` with the values 4 and 9, we get:

$$4 < 3 * 3 = \textit{true}$$
$$9 < 3 * 3 = \textit{false}$$

As expected, we can see that 4 is fair, while 9 is not.

Now that we have looked at the selection step of the explicit SEU model, lets have a look at how the fault instructions work. The modelling of fault instructions can be seen in Figure 17.

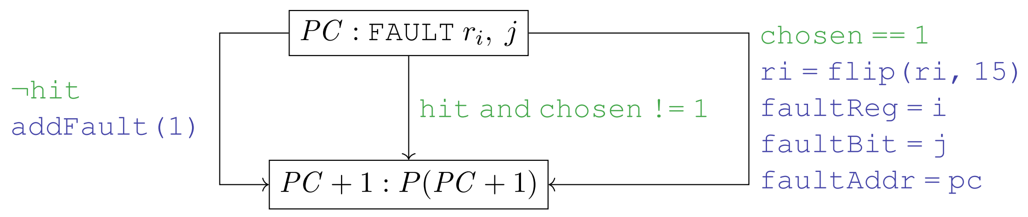


Figure 17: `FAULT` instruction in the explicit SEU model.

When the analysis generates the UPPAAL model, it assigns every fault instruction a number. It is this number that determines whether a fault instruction is executed or not. Figure 17 shows a fault instruction that has been given the number 1. All fault instructions have three outgoing transitions, one is used during the first run and the other two are used during the second run.

The leftmost transition is used by the fault instruction during the first run. This is the run where no faults are executed. Instead we collect all the fault instructions that we encounter. This is done by calling the `addFault` function and passing the fault number. By doing this we store the numbers of all encountered fault instructions in the `visited` array that we use during the selection step.

During the second run, we can no longer use the leftmost transition since `hit` is now true. Instead we choose one of the other transitions depending on which fault instruction was chosen by the selection step that we described earlier. If it is the case that the current fault was not chosen, we simply skip it by using the middle transition. If the fault was chosen, however, we perform the given bitflip.

10 Experiments

In this section we present results of experiments with the analysis toolchain. The quantitative analysis are conducted on two versions of the alarm controller program. The first version is the base program that has been used throughout the report without any fault tolerance measures applied, while the second version has been subjected to software instruction duplication based on the vulnerability analysis.

10.1 Setup

We run the tested programs through the entire analysis pipeline. This means that faults are selected based on the results of the vulnerability analysis. The risk of each fault is then quantified with UPPAAL SMC. We then perform instruction duplication on the faults and run SMC simulations again to check the effect. We present the initial results from the vulnerability analysis (vulnerable and non-equivalent faults) as well as the quantitative results from the statistical model checking.

In order to extract quantitative information from a UPPAAL model, we record a snapshot of the underlying state at appropriate program points. More specifically, we are interested in knowing whether a simulation deadlocked in the *Illegal* location, since this tells us whether a fault broke the security/safety of the program. Additionally, we are also interested in getting information about the activated fault, if any was activated. This is used to build statistics over each individual fault.

The following SMC query collects the above data:

simulate n [#<=s] { CFG.Illegal, CFG.Exit, hit, faultReg, faultBit, faultAddr }

where n is the number of simulations to run and s is the maximum number of transitions to take in each simulation. The value of s will be a number high enough to ensure that the entire program can be traversed.

By running the above query, we get the value of each observed location and variable for each step taken by UPPAAL SMC. Since we are only interested in the final outcome of the simulation, we only collect the values in the final state of the run. By doing this, we can see whether the run ended in the *Illegal* or *Exit* location.

We also record the *hit* variable which tells us whether a fault occurred in the program. This can be used to check how often a fault actually changes the control flow. The last three variables that we record store information about the fault that happened. These variables store the register, bit and address of the fault. We can then run a large number of simulations with the above query to obtain data about each simulation run, which we process to get the results that we are interested in.

10.2 Vulnerability Analysis Results

From the symbolic execution step of the analysis, we determine the faults that can cause illegal control flow. By doing this, we obtain information about which registers and bit positions are vulnerable at given program points. The alarm controller program is shown again in Listing 12.

```
1  LDR r0, 0xFEED   ; Read sensor.
2  MOV r1, 0        ; Min. error bound.
3  MOV r2, 10000    ; Max. error bound.
4  MOV r3, 2000     ; Dangerous temp.
5  CMP r0, r1       ; Check min.
6  BLT noalarm
7  CMP r0, r2       ; Check max.
8  BGT noalarm
9  CMP r0, r3       ; Check danger.
10 BLT noalarm
   alarm:
   ...
11 B exit
   noalarm:
12 ASSERT r0@1 < 2000 ∨ r0@1 > 10000
   ...
   exit:
```

Listing 12: Alarm controller program.

The vulnerability analysis finds the vulnerabilities shown in Table 3.

This table shows the registers that are found to be vulnerable, as well as the bit positions that they are vulnerable on. We can see that register `r0` is vulnerable on numerous bit-positions.

Address	Register	Vulnerable bits
5	`r0`	0-15
5	`r1`	11-14
7	`r0`	0-15
7	`r2`	4, 8-10, 13, 15
9	`r0`	4-15
9	`r3`	0-3, 5, 11-14

Table 3: Vulnerable registers and bit positions identified by vulnerability analysis.

This is likely because `r0` is used for storing the fully symbolic input, and therefore it is more likely that there exist input values that can be flipped to cause unintended program behaviour.

Recall that the vulnerability analysis ensures no equivalent faults. As we saw in Section 8, the vulnerability analysis correctly detects the fault equivalence of the fault immediately before instruction 9 on register `r3` bit 13 and 14 (corresponding to f_{14} and f_{15} in Example 9). This means that the explicit SEU model only contains one of these faults before address 9.

While only two faults were determined to be fault equivalent, we note that a large number of equivalent faults are removed by Step 1 of the vulnerability analysis. Furthermore, we expect that the number of equivalent faults per program increases as more aggressive fault models are considered.

10.3 Quantitative Analysis Experiments

We quantify the risk of each fault in the program by running 200000 simulations with UPPAAL SMC. We first run simulations with the implicit SEU model to establish a baseline probability distribution over faults. We then run simulations with the explicit SEU model to see how they differ from the results of the implicit SEU model. As previously mentioned in Section 9.3, we can control the hit and miss weights of faults. Since we are only interested in the runs where faults occur we use a miss weight of zero to ensure that faults occur in all runs.

Before we present the specific results from each fault model, we first show the overall probability that any fault causes illegal control flow. These probabilities are shown in the blue bars of Figure 18.

We first look at the vulnerability percentages for the base, unhardened program. Here we can see that the probability is more than three times larger in the explicit SEU model compared to the implicit SEU model. This is expected given that all faults in the explicit SEU model will affect the program in some way, while faults in the implicit SEU model may have no effect at all. This is because the implicit model may cause faults to occur on registers that are not live as well as on bit positions that are not vulnerable.

The complete distribution of faults for simulations with the implicit and explicit SEU models are shown in Appendix G and Appendix H. From the data, it is clear that register `r0` is the most vulnerable register. Additionally, we see that the most significant bits are most vulnerable. This is useful information and could be used in combination with a device-specific susceptibility template for physical memory to avoid placing memory pages in blocks, where the vulnerable

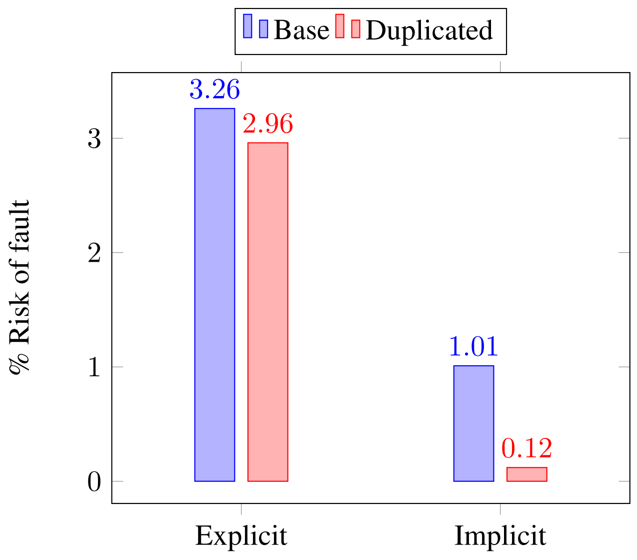


Figure 18: Overall program vulnerability

bit positions are susceptible to bitflips.

In order to easier compare the numbers for implicit and explicit experiments, we now instead look at aggregates of the data.

In Table 4 we present the probability distribution over vulnerable registers and bit positions for the implicit SEU model. Probabilities of zero are not shown.

Bit	**Register**				
	`r0`	`r1`	`r2`	`r3`	
2				0.05%	0.05%
4	0.05%		0.05%		0.10%
5	0.05%			0.05%	0.10%
6	0.10%				0.10%
8	0.30%		0.25%		0.55%
9	0.50%		0.30%		0.79%
10	0.20%		0.89%		1.09%
11	6.40%	0.05%		1.88%	8.33%
12	5.56%	1.59%		4.51%	11.66%
13	12.05%	3.97%	7.24%	8.04%	31.30%
14	10.32%	5.26%		9.62%	25.20%
15	14.29%		6.45%		20.73%
	49.80%	10.86%	15.18%	24.16%	100.00%

Table 4: Distribution of register vulnerabilities for the implicit fault model.

The table shows the frequency for specific registers and bit positions causing vulnerable flow. As we can see, the simulations did not find any vulnerable faults on the bit positions 0, 1, 3, and 7 that were able to break the security. As we saw in Table 3, these bit positions are vulnerable for some of the registers, but after running the simulations, it is apparent that the risk of breaking the security by flipping these bits is very low. However, since the estimated risk is low, the number of path condition models breaking the security is low. To further analyse these rare events, we can leverage the SMT solver to perform "model counting" on the possible satisfying assignments to solutions to the path conditions collected by symbolic execution. While in general, model counting is not efficient, it may be advantageous to use when we know the number of models is small.

We can also see that `r0` and `r3` are the most vulnerable registers, and that it is mainly the most significant bits that are vulnerable.

We have collected similar results for the simulations that use the explicit SEU model. These results can be seen in Table 5. We can see some similarities between the results of the implicit

Bit	**Register**				
	`r0`	`r1`	`r2`	`r3`	
2	0.02%				0.02%
4	0.09%				0.09%
5	0.08%			0.03%	0.11%
6	0.17%				0.17%
7	0.20%				0.20%
8	0.51%		0.12%		0.63%
9	0.38%		0.34%		0.72%
10	0.55%		0.75%		1.30%
11	7.41%	0.02%		1.84%	9.27%
12	7.80%	1.36%		3.38%	12.54%
13	14.20%	4.46%	6.22%	6.13%	31.00%
14	13.14%	6.07%			19.21%
15	18.32%		6.43%		24.75%
	62.86%	11.90%	13.86%	11.38%	100.00%

Table 5: Distribution of register vulnerabilities for the explicit fault model.

and explicit SEU models. It is still the most significant bits that are the most vulnerable and register `r0` is still the most vulnerable register.

One big difference between the results is that the register `r3` now has a significantly lower vulnerability. As we can see, one of the reasons for this is that the implicit SEU model found many vulnerabilities on bit 14, while the explicit SEU model did not find any. This is because the explicit SEU model is not testing SEUs on bit 14 of register `r3`, since it is equivalent with bit 13. This is the downside of removing equivalent faults in the quantitative analysis, as it changes the probabilities. To alleviate this, measures for scaling the probabilities for faults with larger equivalence classes must be developed.

Another difference between the results is that `r0` is now more vulnerable (about the same amount as `r3` is now less vulnerable). One reason for this is that the explicit SEU model does not consider the *windows* in which registers can be flipped. It is reasonable to assume that a register that is live for a long time without being changed is more vulnerable than a register that is live for a short time. In the implicit SEU model this is automatically considered since faults are distributed over the entire program. In the explicit model, however, faults are only encoded before instructions that read registers, and register lifetimes are not considered. This may be the cause of `r0` being much more more vulnerable than the other registers in the explicit SEU model. We consider three different addresses to flip `r0` (before each comparison), while we only consider one address for each of the other registers.

10.4 Quantitative Analysis Experiments with Duplication

In this section we describe how we perform experiments on the version of the alarm controller program in which some instructions have been duplicated to improve fault tolerance. We are using a simple instruction duplication scheme that uses a set of *duplicate registers*, which are used to detect SEUs. The idea behind the duplication scheme is that we compare the original registers to their duplicates before they are used for setting flags. Since it is the flags that control the flow of a program, this scheme reduces the probability of an SEU changing the intended control flow. This complete instruction duplication scheme can be found in Appendix E.

The duplicated version of the alarm controller program can be seen in Appendix F. The program has been generated by finding the vulnerable instructions with symbolic execution and duplicating them as well as all instructions that they have data dependencies on. When using symbolic execution to find vulnerable instructions in the duplicated program, we find that it is the same instructions that are still vulnerable. The newly added comparison instructions cannot break the program's security and are not vulnerable. This means that the duplicated register are not considered by the explicit SEU model, since it only considers vulnerable registers. The implicit SEU model still contains all the duplicated registers to make the fault distribution uniform over all registers.

We are running simulations in UPPAAL SMC in the same way for the duplicated version of the alarm controller program as we did for the base version. This means that we run 200000 simulations on both SEU models with a miss probability of zero.

We start by looking at the overall probability of any fault causing illegal flow for each fault model. These probabilities can again be found in Figure 18 (the red bars).

The overall probability of breaking the security is not much lower for the explicit SEU model on the duplicated code. This is because the model still targets the faults at specific program points just before the vulnerable comparison instructions. Since the vulnerable comparison instructions are located after the fault tolerance instructions, they bypass much of fault tolerance provided by the duplication scheme. On the other hand, the overall probability of illegal control flow for the implicit SEU model is about a tenth of the probability of the original program. This indicates that the simple duplication scheme works better for the randomized faults of the implicit SEU model. One of the reasons for the lower probability is that the model is inducing SEUs on the duplicated registers, which cannot break the security. Another is that the SEUs are not targeted and many faults are caught by the extra fault tolerance comparison instructions. This can also be seen in Table 6, where we can see how many faults are detected by the duplication scheme.

Implicit	**Explicit**
32.25%	15.62%

Table 6: Faults caught by instruction duplication.

We can see that about one third of the faults from the implicit SEU model are detected, since the faults are distributed over the entire program. We cannot detect as many faults in the explicit SEU model, which is expected since it targets the faults *after* the fault tolerance instructions. This means that we can only detect the early faults on `r0`, which are detected by the later comparison instructions.

11 Extension with Loops

We acknowledge the limitations of the TinyARM language, especially the omission of loop constructs due to simplification of the analysis. We now discuss the challenges of extending the analysis to iteration or recursive features.

The simplest form of loop is the statically bounded loop, where the loop condition is bounded by some hard-coded value c. Such loops can simply be unrolled, i.e. transformed into a sequence of the loop body instructions repeated c times. Preprocessing such loops beforehand, enables the analysis to proceed without modification.

A more problematic type of loop is the unbounded loop, also called an infinite loop. Such loops cannot readily be unrolled, and it is not clear how the analysis should proceed. Furthermore, the current fault instruction formalization is not suitable for explicitly modelling faults occurring after some specific number of iterations in a loop.

A loop variant of the previously presented alarm controller program is shown in Figure 19. The loop version of the alarm program is conceptually similar to the program we have previously seen. Whether the alarm is triggered depends on the state of the `unsafe` variable. If the program detects a non erroneous temperature over the safety threshold of 2000, the `unsafe` variable is set to 1 and the alarm is triggered when control is transferred outside the loop body. Similar to the loop-free alarm controller program, we have two safety cases:

- *False positives*: the alarm is triggered but the actual temperature reading is not unsafe.
- *False negatives*: the actual temperature reading is indeed unsafe, but for some reason the alarm is not triggered.

While false positives can disrupt normal workflow of the users of the system, the consequences of false negatives are potentially catastrophic.

Notice that the loop condition is unbounded. This is problematic from an analysis perspective, since the symbolic execution step of the vulnerability analysis must explore a possibly

```c
short unsafe = 0;
while (!unsafe) {
  short temp = sensor(0xFEED);
  if (temp < 10000 &&
      temp >= 2000) {
    unsafe = 1;
  }
}
alarm();
```

```
MOV   r3, 0        ; unsafe = 0
loop:
MOV   r4, 0
CMP   r3, r4
BNE   alarm
LDR   r0, 0xFEED   ; Read sensor.
MOV   r1, 10000    ; Max.
CMP   r0, r1       ; Check max.
BGE   loop
MOV   r2, 2000     ; Danger temp.
CMP   r0, r2       ; Check danger.
MOVGE r3, 1        ; unsafe = 1
B loop
alarm:
```

Figure 19: Alarm controller program with loop.

infinite number of traces to the target instruction. A popular approach for dealing with this problem is to bound the symbolic execution to some maximal depth in the loop. Alternatively, one can use a concrete execution trace to guide the search. This is known as concolic testing[20].

For the above alarm controller program, we observe that one iteration of the loop body does not affect future iterations of the loop body statements. In other words, there are no circular data dependencies between instructions in the loop body. Additionally, the path conditions on the variables in the loop body are invariant after the first iteration of the loop. With these observations in mind, we propose to transform the loop into a single sequential execution of the loop body, and move the check on the unsafe variable to the end of the instructions. Essentially, we unroll one representative iteration of the loop and check for safety after. The unrolled version of the program can be seen in Listing 13.

```c
short unsafe = 0;
short temp = sensor(0xFEED);
if (temp < 10000 &&
    temp >= 2000) {
  unsafe = 1;
}
if (unsafe)
  alarm();
```

Listing 13: Single iteration of alarm controller with loop.

Since temperature readings in the loop body are fully symbolic, we get the same set of models from solving the path conditions for the sequential transformation as we would from the loop. The path conditions to the `alarm` call for both variants of the program can be found in Table 7. In essence, this is possible since we can obtain a fixed-point on the path conditions generated by symbolic execution. The transformation enables us to proceed with the analysis as before, without further modification.

In Listing 14 we see a dynamically bounded robot actuator program.

Program	Conditions
Figure 19	$(s_0 < 10000 \land s_0 \geq 2000 \land 1 \neq 0) \lor$ $((s_0 \geq 10000 \lor s_0 < 2000) \land 0 \neq 0)$
Listing 13	$(s_0 < 10000 \land s_0 \geq 2000 \land 1 \neq 0) \lor$ $((s_0 \geq 10000 \lor s_0 < 2000) \land 0 \neq 0)$

Table 7: Path conditions to `alarm()` after one iteration.

```
short n = read(0x1234);
short i = 0;
while (i < n) {
    Move();
    i++;
}
```

Listing 14: Dynamically bounded loop.

The program takes as input a number of steps `n` and moves some robot arm `n` units. Observe that the loop guard depends on the increment operation in the loop body, and the increment operation depends on itself from the previous iteration. Since a fault during any iteration of the loop may not yield the same path conditions on the variables, we cannot apply the proposed sequential transformation to unroll the loop.

We argue that dynamically bounded loops should be avoided in safety/security critical systems, since they are difficult to analyse and we estimate these loops to be especially susceptible to bitflip faults since they depend on symbolic input, which we observed in the experiments on the alarm controller program to be very vulnerable.

Lastly, we note that classifying the type of the loops in a program during analysis could be done by checking the Program Dependency Graph for circular dependencies.

12 Conclusion and Future Work

We have formalised a small assembly language based on ARM, with explicit syntax and semantics for modelling SEU faults. We have formalised and implemented a formal static analysis for detecting the vulnerability of faults encoded in programs with respect to a critical program point. The analysis consists of data and control flow analyses to obtain a program slice to the critical program point, and it only considers faults along traces in the slice.

For the analysis, we have implemented a symbolic execution engine that collects path conditions generated along the traces to the critical program point. By solving the path conditions with an SMT solver, we determine which faults can break the safety/security of a program, thus causing vulnerabilities in the system.

We have further presented a formal definition of fault equivalence and proposed an algorithm for determining the equivalence of faults. We have shown how fault equivalence testing can be used to reduce the number of faults needed for analysis and highlighted other benefits of analysing fault equivalence classes rather than individual faults.

Furthermore, we have shown how TinyARM programs can be encoded as timed automata, enabling modelling and analysis of programs by statistical model checking.

Finally, we have implemented an analysis toolchain and show how the analysis can be used to quantify the risk of faults.

Future Work

Since the analysis we have presented works on an idealized ARM variant, several extensions must be made to enable analysis of real ARM binaries.

An important modification to the quantitative analysis would be to consider the lifespan of registers in the explicit SEU model. Currently, it does not differentiate between registers that are live for a single instruction and registers that are live for the entire duration of the program. This is important to consider when calculating probabilities, since registers with a longer lifespan are more likely to be hit by SEUs. This is not a problem when using the implicit SEU model, but since it uses unnecessary time testing faults that are not vulnerable, it is more interesting to encode it into the explicit SEU model. This would require some metrics that can be used with the results to normalise them according the registers' lifespans.

One extension that can be made to the analysis is to extend it to support SEUs in other parts of a program, aside from the registers. A simple extension would be to look at SEUs in memory and flags. Flags are straightforward to consider SEUs in, since flags are just simple 1-bit registers. As for memory, it is possible to look at the content of memory addresses and induce faults into this content. This would be similar to the current analysis, since memory addresses can interpreted similar to registers. This means that SEUs could occur at given addresses, which would result in the stored data changing value.

While the number of equivalent faults found in our experiments were low, fault equivalence could be extended to include SEUs in memory and flags as well. This should provide more interesting equivalence classes of faults. The current equivalence definition should already be able to handle SEUs in memory, since memory can affect the content of the conditional flags through registers. Some faults on registers could therefore be equivalent to faults in some memory locations, since the memory is loaded into given registers before it is used. However, modifications to the equivalence definition are needed in order to handle SEUs in flags. Currently the definition requires flags at all program points to be the same for two faults. This will cause problems when the faults are not flipping the same flag at the same address, since some intermediate addresses will have different flags. To overcome this, the definition could be modified to only require the flags to be the same at program points where conditional execution happens, e.g. `MOVGT` or `BLT`. For all program points with unconditional execution, the flags do not have any effect anyway and we can therefore relax this condition on the definition of fault equivalence.

For more aggressive fault models such as SEUs in the control register or the instruction encoding it becomes more complex how to extend the analysis. These types of faults are harder to analyse since they can severely modify the behaviour of programs. This means that the program representations used by the analysis need to capture these possible behaviours. One way of doing this is to consider the possible faults in the CFG. For faults in the program counter, this would mean that the vertices of the CFG would get more outgoing edges to capture the fact that the faults can cause jumps to other parts of the program. Faults in the instruction encoding can also be represented in the CFG. Such faults would cause the CFG to contain more vertices, which capture the different ways that the original instructions can be mutated. A big issue with these kinds of faults, in the current analysis, is that they can change the flow of the program to jump backwards and thereby introduce loops. Because of this, the analysis also needs to be extended to handle loops before it can fully handle faults in the instruction encoding. However, under a standard SEU fault model, we conjecture that the 'loop' induced by a fault in the control register or instruction encoding of jump targets should result in a loop that can be handled by the transformations discussed in Section 11.

The current fault equivalence definition assumes that the two programs are executing the same program. This would no longer necessarily be the case when we consider faults in the instruction encoding. These faults may change the semantics of the program and the definition would not be able to correctly identify equivalent faults. For such faults, it may be needed to develop a entirely different view on fault equivalence classes. In this case, it may be useful to leverage techniques from for detecting bisimilarity of programs[1].

Since statistical model checking requires a large amount of simulations when analysing rare events, it would be interesting to explore other ways of calculating the probabilities.

One way of calculating the probabilities could be to look at fault equivalence, and examine whether it is a viable option for probability calculation. We could encode all possible faults into a program and have faults for every register and bit position between all the original program instructions. After doing this, we can find the equivalence classes for all the faults. This would result in a number of equivalence classes with different cardinalities. Then by using these cardinalities, we can calculate the probability that a given *fault effect* happens. This could possibly be used as a quicker way of doing the quantitative analysis.

Another way to calculate the probabilities could be to get the exact probabilities through symbolic execution and SMT solving. To do this, we could count all models for a vulnerable register or bit position and thereby get the precise probabilities. While counting SMT models is a hard program, we can leverage the fact that our SMT formulas are finite. This means that they can be transformed into equivalent SAT formulas, which can be counted by using a #SAT solver such as sharpSAT[26].

References

[1] ACETO, L., INGÓLFSDÓTTIR, A., LARSEN, K. G., AND SRBA, J. *Reactive systems: modelling, specification and verification.* cambridge university press, 2007.

[2] ANDERSEN, T. R., AND TERNDRUP, M. K. Bitflip Analysis of an ARM-based Assembly Language. Unpublished, January 2018.

[3] ARM LIMITED. *ARM Architecture Reference Manual*, July 2005. Issue I.

[4] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 463–469.

[5] DAVID, A., LARSEN, K. G., LEGAY, A., MIKUČIONIS, M., AND WANG, Z. Time for statistical model checking of real-time systems. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 349–355.

[6] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, pp. 337–340.

[7] FENG, S., GUPTA, S., ANSARI, A., AND MAHLKE, S. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 385–396.

[8] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[9] HANSEN, R. R., LARSEN, K. G., OLESEN, M. C., AND WOGNSEN, E. R. Formal modelling and analysis of Bitflips in ARM assembly code. *Information Systems Frontiers 18*, 5 (Oct. 2016), 909–925.

[10] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 361–372.

[11] KISS, A., JASZ, J., LEHOTAI, G., AND GYIMOTHY, T. Interprocedural static slicing of binary executables. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation* (Sept 2003), pp. 118–127.

[12] MAY, T. C., AND WOODS, M. H. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices 26*, 1 (Jan 1979), 2–9.

[13] MEOLA, M. L., AND WALKER, D. Faulty logic: reasoning about fault tolerant programs. In *European Symposium on Programming* (2010), Springer, pp. 468–487.

[14] MØLLER, A., AND SCHWARTZBACH, M. I. Static Program Analysis. `https://cs.au.dk/~amoeller/spa/`, September 2017. Accessed: 18-12-2017.

[15] MUCHNICK, S. S. *Advanced compiler design implementation.* Morgan Kaufmann, 1997.

[16] MUNKBY, G., AND SCHUPP, S. Type inference for soft-error fault-tolerance prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), IEEE Computer Society, pp. 65–75.

[17] PERRY, F., AND WALKER, D. Reasoning about control flow in the presence of transient faults. In *International Static Analysis Symposium* (2008), Springer, pp. 332–346.

[18] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 1–18.

[19] Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2005), CGO '05, IEEE Computer Society, pp. 243–254.

[20] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 317–331.

[21] Seaborn, M., and Dullien, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, March 2015. Accessed: 11-12-2017.

[22] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS* (2015).

[23] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 138–157.

[24] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[25] Tang, A., Sethumadhavan, S., and Stolfo, S. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1057–1074.

[26] Thurley, M. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006* (Berlin, Heidelberg, 2006), A. Biere and C. P. Gomes, Eds., Springer Berlin Heidelberg, pp. 424–429.

[27] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1675–1689.

A Updating Conditional Flags

This appendix is from our previous work[2]. It has been updated to use 16-bit values.

This appendix contain the functions that are used to update the condition flags when they are set by `ADDS`, `SUBS`, `MULS`, and `CMP`.

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_N) = \begin{cases} 1 & \text{if } v_1 + v_2 \geq 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_Z) = \begin{cases} 1 & \text{if } v_1 + v_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_C) = \begin{cases} 1 & \text{if } v_1 + v_2 > 2^{16} - 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_V) = \begin{cases} 1 & \text{if } v_1, v_2 < 2^{15} \wedge (v_1 + v_2 \geq 2^{15}) \\ 1 & \text{if } v_1, v_2 \geq 2^{15} \wedge (v_1 + v_2 < 2^{15}) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_N) = \begin{cases} 1 & \text{if } v_1 - v_2 \geq 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_Z) = \begin{cases} 1 & \text{if } v_1 - v_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_C) = \begin{cases} 1 & \text{if } v_1 - v_2 < 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_V) = \begin{cases} 1 & \text{if } v_1 < 2^{15} \wedge v_2 \geq 2^{15} \wedge (v_1 - v_2 \geq 2^{15}) \\ 1 & \text{if } v_1 \geq 2^{15}0 \wedge v_2 < 2^{15} \wedge (v_1 - v_2 < 2^{15}) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_N) = \begin{cases} 1 & \text{if } v_1 * v_2 \geq 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_Z) = \begin{cases} 1 & \text{if } v_1 * v_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_C) = F(f_C) \quad \texttt{MULS} \textit{ does not affect the carry flag.}$$

$$\mathit{flags}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_V) = F(f_V) \quad \texttt{MULS} \textit{ does not affect the overflow flag.}$$

$$\mathit{flags}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_N) = \begin{cases} 1 & \text{if } v_1 - v_2 \geq 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_Z) = \begin{cases} 1 & \text{if } v_1 - v_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_C) = \begin{cases} 1 & \text{if } v_1 - v_2 < 2^{15} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{flags}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_V) = \begin{cases} 1 & \text{if } v_1 < 2^{15} \wedge v_2 \geq 2^{15} \wedge (v_1 - v_2 \geq 2^{15}) \\ 1 & \text{if } v_1 \geq 2^{15}0 \wedge v_2 < 2^{15} \wedge (v_1 - v_2 < 2^{15}) \\ 0 & \text{otherwise} \end{cases}$$

B Updating Symbolic Conditional Flags

This appendix is from our previous work[2]. It has been updated to use 16-bit values.

This appendix contain the functions that are used to update the condition flags during symbolic execution, when they are set by `ADDS`, `SUBS`, `MULS`, and `CMP`.

$$
\begin{aligned}
\mathit{flagsSe}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_N) &= v_1 + v_2 \geq 2^{15} \\
\mathit{flagsSe}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_Z) &= v_1 + v_2 = 0 \\
\mathit{flagsSe}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_C) &= v_1 + v_2 > 2^{16} - 1 \\
\mathit{flagsSe}_{\texttt{ADD}}(v_1,\ v_2,\ F)(f_V) &= ((v_1 > 0 \land v_2 > 0 \land (v_1 + v_2 \geq 2^{15})) \\
&\quad \lor (v_1 < 0 \land v_2 < 0 \land (v_1 + v_2 \geq 0)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flagsSe}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_N) &= v_1 - v_2 \geq 2^{15} \\
\mathit{flagsSe}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_Z) &= v_1 - v_2 = 0 \\
\mathit{flagsSe}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_C) &= v_1 - v_2 < 2^{15} \\
\mathit{flagsSe}_{\texttt{SUB}}(v_1,\ v_2,\ F)(f_V) &= ((v_1 > 0 \land v_2 < 0 \land (v_1 - v_2 \geq 2^{15})) \\
&\quad \lor (v_1 < 0 \land v_2 > 0 \land (v_1 - v_2 \geq 0)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flagsSe}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_N) &= v_1 * v_2 \geq 2^{15} \\
\mathit{flagsSe}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_Z) &= v_1 * v_2 = 0 \\
\mathit{flagsSe}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_C) &= F(f_C) \quad \texttt{MULS}\ \textit{does not affect the carry flag.} \\
\mathit{flagsSe}_{\texttt{MUL}}(v_1,\ v_2,\ F)(f_V) &= F(f_V) \quad \texttt{MULS}\ \textit{does not affect the overflow flag.}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flagsSe}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_N) &= v_1 - v_2 \geq 2^{15} \\
\mathit{flagsSe}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_Z) &= v_1 - v_2 = 0 \\
\mathit{flagsSe}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_C) &= v_1 - v_2 < 2^{15} \\
\mathit{flagsSe}_{\texttt{CMP}}(v_1,\ v_2,\ F)(f_V) &= ((v_1 > 0 \land v_2 < 0 \land (v_1 - v_2 \geq 2^{15})) \\
&\quad \lor (v_1 < 0 \land v_2 > 0 \land (v_1 - v_2 \geq 0)))
\end{aligned}
$$

C Control Flow Graph Extraction

This appendix is from our previous work[2].

This section describes the transformation rules that we apply to every TinyARM instruction in order to transform a TinyARM program into a CFG.

We first present the transformation rules for unconditional TinyARM instructions, which can be found in Figure 20. When building the CFG, these rules will be applied to every unconditional

$$\begin{tikzcd}
\boxed{PC : \texttt{I}} \arrow[d] \\
\boxed{PC + 1 : P(PC + 1)}
\end{tikzcd}$$

(a) Unconditional sequential instruction.
$\texttt{I} \in \{\texttt{MOV}, \texttt{LDR}, \texttt{CMP}, \texttt{OP}, \texttt{OPS}\}$.

$$\begin{tikzcd}
\boxed{PC : \texttt{B}\ a} \arrow[d] \\
\boxed{a : P(a)}
\end{tikzcd}$$

(b) Unconditional branch instruction.

Figure 20: CFG transformation rules for unconditional instructions.

instruction in the TinyARM program. The unconditional transformation rules for sequential instructions are not very complicated, they simply add an edge between the current instruction and the next instruction (Figure 20a). The branch transformation rule just adds an edge from the branch instruction to its target, instead of adding an edge to the next instruction (Figure 20b).

The transformation rules for conditional TinyARM instructions are presented in Figure 21. These rules will be applied to every conditional instruction of a TinyARM program, when the

$$\begin{tikzcd}
\boxed{PC : \texttt{I}\chi} \arrow[d] \\
\boxed{PC + 1 : P(PC + 1)}
\end{tikzcd}$$

(a) Conditional sequential instruction.
$\texttt{I} \in \{\texttt{MOV}, \texttt{LDR}, \texttt{CMP}, \texttt{OP}, \texttt{OPS}\}$.

$$\begin{tikzcd}
& \boxed{PC : \texttt{B}\chi\ a} \arrow[dl] \arrow[dr] & \\
\boxed{a : P(a)} & & \boxed{PC + 1 : P(PC + 1)}
\end{tikzcd}$$

(b) Conditional branch instruction.

Figure 21: CFG transformation rules for conditional instructions.

CFG is being build. The transformation rules for conditional TinyARM instructions are a little different. A conditional instruction will only be executed if its condition is true. If it is not, the instruction will act as a `NOP` instruction and not do anything, except incrementing the program counter. Conditional sequential instructions will look the same as unconditional instructions in the CFG (Figure 21a). However, the instruction can have two different executions depending on its condition code. We will handle the different executions during the analysis of the instruction (if it is necessary for the specific analysis). The conditional branch acts a little different from the unconditional branch. If the branch executes, it will jump to the program point at the branch target, but if it does not execute it will just continue to the next instruction, like the sequential instructions do (Figure 21b).

Together these transformation rules allow us construct CFGs that can represent any TinyARM program.

D Alarm Controller Program in UPPAAL

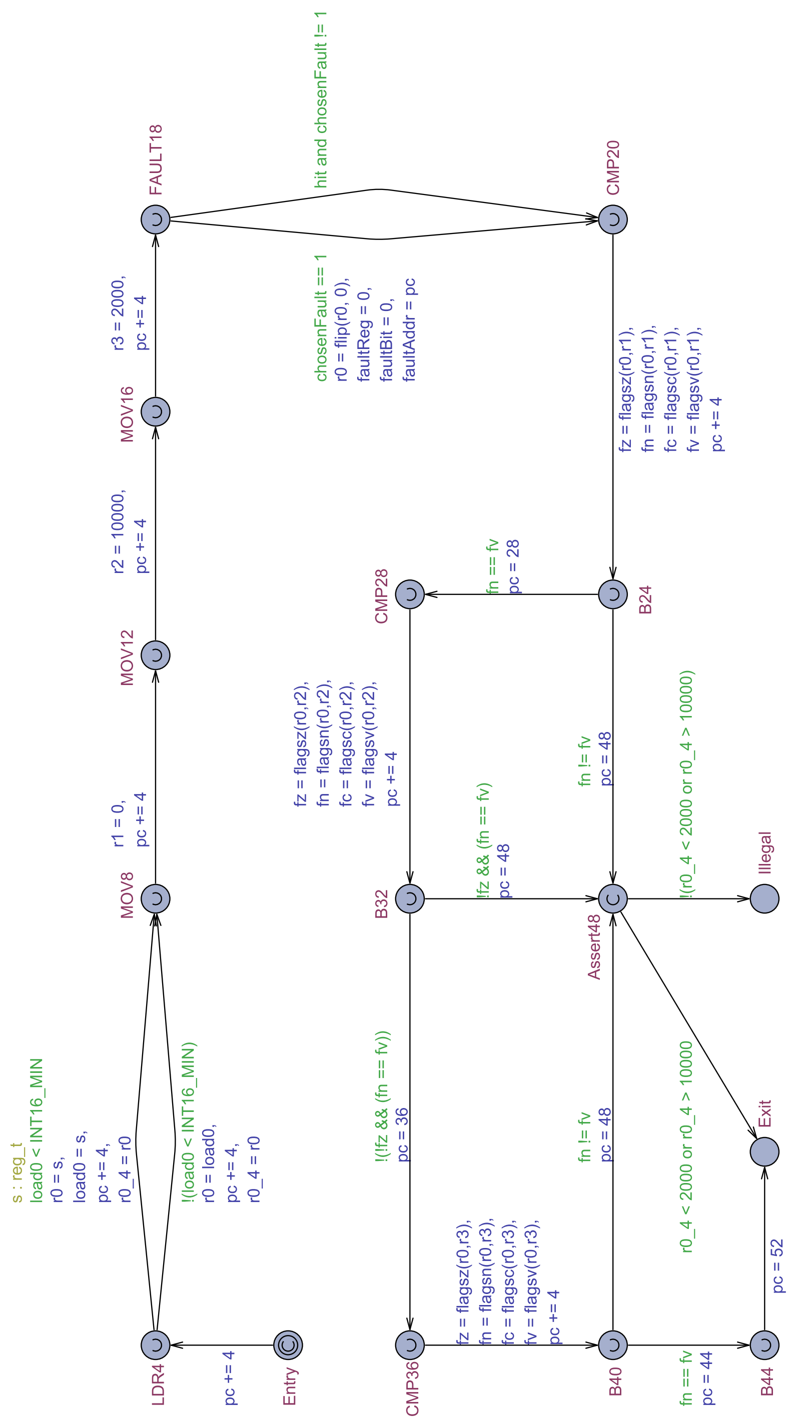


Figure 22: UPPAAL model of alarm controller program.

E Instruction Duplication

In this section we describe how we use the results of our analysis to duplicate the instructions of programs in order to make them less vulnerable to SEUs.

Duplicating instructions is a common way to improve the fault tolerance of programs. Instruction duplication involves executing instructions twice and comparing their outcomes. This allows one to check if a fault has happened in one of the instructions, since this would result in different outcomes between the instructions.

The obvious disadvantage of instruction duplication is that the program will run slower, since it needs to execute instructions twice. Furthermore, extra memory will also be needed in order to store the values of both the original registers and their copies. While some static program analyses can help alleviate these problems, e.g. by optimising memory reusage, it still makes sense to identify the critical instructions and only duplicate them. This can improve on the program performance compared to duplicating all instructions.

We are working with an ARM-based language which means that we will have to consider how the execution of instructions affects the rest of the program. This is the case since some instructions change the conditional flags which control the future execution of the program.

As mentioned at the start of this section, we will be using the results of our analysis to identify the instructions that should be duplicated. By using the analysis we do not have to duplicate all instructions, since some instructions cannot change the control flow to reach critical code. The procedure for finding the instructions that should be duplicated works in the following way:

1. Find all faults that can cause control flow to reach critical code (given by the analysis).
2. From each of these faults: Find the next instruction, i, that reads the faulty register (this will usually be an instruction shortly after the fault).
3. From each i perform a backwards search in the DDG to find the set of all instructions, I, that affects i in some way ($i \in I$).
4. Duplicate all the instructions in the I sets.

The way we duplicate an identified instruction is by adding a copy of it immediately after the original instruction. This copy will be using separate registers, such that we can compare the values of the original and copied registers when we use them later in the program execution. The only places we will be comparing the duplicated registers are when they are used to set the conditional flags. In other words we only compare duplicated registers when they are used in `CMP`, `ADDS`, `SUBS`, or `MULS`. It is enough to compare the duplicated registers at these points, since these instructions are the only ones that can change the program's control flow. Furthermore, instructions that do not use any registers (only branches) will not be duplicated, since they cannot change the program's behaviour by themselves.

We will now show how the different instructions are duplicated and explain some of the choices we have made in regards to instruction duplication.

First, we show how the instructions that do not affect conditional flags have been duplicated. This includes the following instructions: `MOV`, `LDR`, `ADD`, `SUB`, and `MUL`. Listing 15 and Listing 16 show how an ADDχ instruction is duplicated. The duplication rules for the other instructions are identical.

```
...
ADDχ r1, r2, r3
...
```

Listing 15: Before ADDχ duplication.

```
...
ADDχ r1, r2, r3
ADDχ r1', r2', r3'
...
```

Listing 16: After ADDχ duplication.

We can see that the instruction is duplicated by placing a copy of instruction between it and the next instruction. This means that the duplicated instructions will just be a part of the complete program with the only distinction being that they use separate registers. These duplicated registers will then be used when we encounter an instruction that sets the conditional flags.

The instructions that set the conditional flags are: CMP, ADDS, SUBS, and MULS. These instructions are not duplicated in the same way as the other instructions, since they will also include a *synchronization* step that compares the registers they use with the duplicated registers. When comparing the registers we end up updating the conditional flags of the program in a way that did not happen in the original program. This, however, is not an issue since these instructions will update the conditional flags again and thereby resume normal program behaviour. There will still be a problem when doing synchronization, since the MULS instruction does not update all conditional flags. We will get back to this issue later in the section.

In Listing 17 and Listing 18 we show how a flag setting instruction (except MULS) is duplicated.

```
...
ADDSχ r1, r2, r3
...
```

Listing 17: Before ADDSχ duplication.

```
...
Bχ execute
B skip
execute:
CMP r2, r2'
BNE fail
CMP r3, r3'
BNE fail
ADDS r1, r2, r3
ADD r1', r2', r3'
skip:
...
```

Listing 18: After ADDSχ duplication.

As we can see, Listing 18 contains a lot of instructions that Listing 17 does not have. The changes that have been made to the duplicated code are:

1. The condition code, χ, has been moved from the ADDSχ instruction to a branch at the start. This means that we only perform the synchronization if the ADDSχ is going to be executed. If χ is not true, we skip everything. This step can be skipped if the condition code is AL.

2. Synchronization of the operands (r2 and r3) is performed before the ADDS instruction. If any of the operands are different from their duplication the program fails.

3. A copy of the ADDS instruction has been added after the original. The copy is just a normal ADD instruction, since we do not need to update the conditional flags twice.

The duplication of SUBS is the same as the duplication of ADDS. The duplication of CMP is mostly the same, the only difference being that the duplication of CMP does not contain the copy in Line 9. The reason for this is that a copy of CMP would just end up setting the flags twice to the same values.

The duplication of MULS is similar to the duplication of the other flag setting instructions. The difference is that we cannot use CMP instructions to do the synchronization. The reason for this is that MULS only updates the negative and zero flags while leaving the carry and overflow flags unaffected. If we use a CMP instruction to synchronize before the MULS instruction, we would change the carry and overflow flags from their intended values and the MULS instruction would not change them back. The way we overcome this issue is by doing the synchronization with SUB and MULS instructions, as this will leave the carry and overflow flags with their original values. This duplication scheme can be seen in Listing 19 and Listing 20

```
...
MULSχ r1, r2, r3
...
```

Listing 19: Before MULSχ duplication.

```
1  ...
2  Bχ execute
3  B skip
   execute:
4  SUB r20, r2, r2'
5  MOV r21, 1
6  MULS r20, r20, r21
7  BNE fail
8  SUB r20, r3, r3'
9  MOV r21, 1
10 MULS r20, r20, r21
11 BNE fail
12 MULS r1, r2, r3
13 MUL r1', r2', r3'
   skip:
14 ...
```

Listing 20: After MULSχ duplication.

We can see that the synchronization now uses SUB and MULS as well as two dummy registers, r20 and r21, instead of using a CMP instruction. This works in the following way:

1. After the SUB instruction in Line 4, the value of r20 will be 0 if r2 = r2'.

2. The MULS instruction in Line 6 will then multiply r20 by 1, and the result of this multiplication will only be 0 if r20 was already 0. This means that the zero flag will be set if r2 = r2'.

Since BNE is executed if the zero flag is not set, we can use the above method to compare registers for equality without updating the carry and overflow flags. This way of synchronizing requires a few extra instructions, but we are not affecting the carry and overflow flags, which means that the program will have these flags set to their intended values.

F Alarm Controller Program with Instruction Duplication

```
LDR r0, 0xFEED
LDR r0x, 0xFEED
MOV r1, 0
MOV r1x, 0
MOV r2, 10000
MOV r2x, 10000
MOV r3, 2000
MOV r3x, 2000
CMP r0, r0x
BNE fail
CMP r1, r1x
BNE fail
CMP r0, r1
BLT noalarm
CMP r0, r0x
BNE fail
CMP r2, r2x
BNE fail
CMP r0, r2
BGT noalarm
CMP r0, r0x
BNE fail
CMP r3, r3x
BNE fail
CMP r0, r3
BLT noalarm
B exit
noalarm:
ASSERT r0@1 < 2000 ∨ r0@1 > 10000
B exit
fail:
B exit
exit:
```

Listing 21: Alarm controller program with instruction duplication.

G Fault Distribution Implicit

A	**R**	**Bit**												
		2	4	5	6	8	9	10	11	12	13	14	15	
8	`r0`				0.05%		0.15%	0.05%	1.29%	0.79%	1.19%	1.64%	1.24%	6.40%
12	`r0`						0.05%		0.89%	0.79%	2.13%	1.49%	1.88%	7.24%
	`r1`									0.50%	1.69%	1.64%		3.82%
16	`r0`		0.05%			0.05%	0.15%	0.10%	1.04%	0.89%	2.18%	1.79%	1.98%	8.23%
	`r1`									0.45%	0.94%	1.69%		3.08%
	`r2`						0.05%	0.25%			1.79%		1.19%	3.27%
20	`r0`			0.05%		0.20%	0.05%		0.79%	0.79%	1.74%	1.74%	2.23%	7.59%
	`r1`								0.05%	0.64%	1.34%	1.93%		3.97%
	`r2`		0.05%			0.10%	0.15%	0.20%			1.54%		1.79%	3.82%
	`r3`								0.35%	0.84%	1.59%	2.03%		4.81%
24	`r0`					0.05%	0.05%		0.84%	0.94%	2.18%	1.44%	1.98%	7.49%
	`r2`					0.10%		0.25%			1.98%		1.79%	4.12%
	`r3`								0.30%	0.94%	1.88%	1.93%		5.06%
28	`r0`						0.05%	0.05%	0.74%	0.84%	1.98%	2.23%	1.34%	7.24%
	`r2`					0.05%	0.10%	0.20%			1.93%		1.69%	3.97%
	`r3`			0.05%					0.25%	0.99%	1.24%	1.98%		4.51%
32	`r0`				0.05%				0.35%	0.25%	0.45%		1.69%	2.78%
	`r3`								0.25%	0.84%	1.49%	1.74%		4.32%
36	`r0`								0.45%	0.25%	0.20%		1.93%	2.83%
	`r3`	0.05%							0.74%	0.89%	1.84%	1.93%		5.46%
		0.05%	0.10%	0.10%	0.10%	0.55%	0.79%	1.09%	8.33%	11.66%	31.30%	25.20%	20.73%	100.00%

Table 8: Fault distribution for implicit SEU model.

H Fault Distribution Explicit

A	**R**	**Bit**												
		2	4	5	6	7	8	9	10	11	12	13	14	15
20	`r0`		0.05%	0.02%	0.09%	0.08%	0.26%	0.17%	0.31%	2.89%	3.23%	6.49%	6.42%	6.37%
	`r1`									0.02%	1.36%	4.46%	6.07%	
28	`r0`	0.02%	0.03%	0.03%	0.08%	0.06%	0.18%	0.18%	0.25%	3.08%	3.00%	6.42%	6.72%	5.84%
	`r2`						0.12%	0.34%	0.75%			6.22%		6.43%
36	`r0`		0.02%	0.03%		0.06%	0.06%	0.03%		1.44%	1.56%	1.29%		6.11%
	`r3`			0.03%						1.84%	3.38%	6.13%		
		0.02%	0.09%	0.11%	0.17%	0.20%	0.63%	0.72%	1.30%	9.27%	12.54%	31.00%	19.21%	24.75%

Table 9: Fault distribution for explicit SEU model.