Review of Methods for Handling Bitflips using Formal Fault Models

Pre-specialisation

Group:
SW908E19

Supervisor:
René Rydhof Hansen

January 15, 2020

AALBORG UNIVERSITY
STUDENT REPORT

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

Title:
Review of Methods for Handling Bitflips using Formal Fault Models

Theme:
Pre-specialisation

Project Period:
Fall 2019

Group:
SW908E19

Participants:
Anton Christensen
Henrik H. Sørensen

Supervisor:
René Rydhof Hansen

Pages:
29

Date of Completion:
January 15, 2020

Number of Copies:
1

Abstract:

A plethora of techniques exist for detecting and correcting errors that can occur in a CPU when it is subjected to a bitflip. In this paper we review a number of such techniques and attempt to categorise them based on the type of errors that can occur. We do based on a formal definition of the errors, a so-called fault model, which is defined on the basis of a small language, TinyARM, with well defined semantics that aim to model a simplified ARM processor. We find that it is difficult to compare them like this, because the techniques are not designed for TinyARM and because they are differ on too many other areas than fault models alone. Regardless, we conclude that if new techniques were developed around a common framework it would be beneficial for comparing them. Additionally, we exemplify how a technique can be formalised through the help of a language that was designed for another technique. We do so by using structured formal proofs to strengthen the argumentation of Moro et al., 2014 while also showing how this structured approach can reduce the risk of making mistakes by showing how a single instruction was classified erroneously by the original authors.

[illegible]

Anton Christensen
achri15@student.aau.dk

[illegible]

Henrik H. Sørensen
hsaren14@student.aau.dk

Contents

1 Introduction

Protecting various microprocessors against a Single Event Upset (SEU), colloquially known as a bitflip, has been the focus of much research over the years. Initially, the focus was on systems for use in space and high altitude aircraft, as those environments contain a higher number of cosmic rays compared to ground level, which significantly increases the probability of one inducing a transient fault, i.e. a fault that does not permanently damage the hardware (Tront, Armstrong, and Oak, 1985). More recently, the search for increases in processor performance has led to ever smaller transistors using lower voltages. Much like putting hardware in space, increasing the density of its internals carries with it a greater risk of transient faults (Shivakumar et al., 2002; Borkar, 2005).

The research into protecting against SEUs is as varied as the types of SEUs: depending on where in a microprocessor an SEU occurs, it can have vastly different effects on a running program. Consider, for example, the following piece of ARM-like assembly code, written in TinyARM, a language we define formally in this paper:

```
1    ldr r1, 0xBAADC0DE  ; move entered PIN from memory to r1
2    mov r2, 3295        ; move correct code into r2
3    cmp r1, r2          ; compare the values in r1 and r2
4    b_ne wrongPin       ; branch to wrongPin if values differed
```

Imagine a machine with memory mapped input/output, where a connected keypad stores its input at the memory address `0xBAADC0DE`. In the code above, the input is moved into register `r1` and the hard-coded correct PIN `3295` is moved to register `r2`. When the two are compared, they change the values of control flags depending on the result of the comparison. In the fourth line control is transferred to a different region of code if the zero flag is not set, i.e. the two values were not equal. Now consider the different ways a single bit being flipped could lead to an erroneous result, with some of them even permitting access without entering the correct PIN. If an SEU occurs in either register `r2` or `r2` after they have been set but before they are compared, the program will deny a correct PIN or might accept an incorrect PIN. The latter is significantly less likely to happen, but if the SEU instead occurs in the zero control flag between lines 3 and 4, both outcomes are equally probable. If an SEU can cause the program counter to change, it is also possible to skip entire instructions. If the third line is skipped, the conditional branching in the next line is dependent on whatever the flags have previously been set to, while skipping the fourth line would simply skip the PIN check entirely. A single instruction can also be skipped, if it is subjected to an SEU that changes its opcode, thereby transforming it into a different instruction entirely, which might not have any impact on the flags or registers it should.

Because of the varying ways SEUs can impact a given program, it is important to clearly convey which types of faults a potential solution can protect against and which it cannot. This information is known as a fault model, and ideally all research into protection against SEUs should be very clear about which fault model is targeted. However, as a cursory literature overview reveals, this is often not the case.

While efforts have been made to define fault models in a rigorous manner such that solutions can be formally proven (Perry et al., 2007; Hansen et al., 2016), most research is based on an often somewhat vague, textual description of the fault model and proving its efficacy through benchmarks. In this paper we attempt to compare a range of different approaches to fault tolerance systematically, by constructing semantic rules that unambiguously model the fault model in a language of our choosing, a redefinition of the TinyARM language from Hansen et al., 2016. While we concentrate on software-only approaches, a few of them make guarantees about their solution based on hardware support. In these cases, we attempt to describe their fault model through our TinyARM semantic rules such that their hardware changes are captured in the fault model. Additionally, we attempt to provide a more rigorous proof of the method presented by Moro et al., 2014 by using semantic rules to show how TinyARM instructions can be made fault tolerant, either by repeating them or by rewriting them to a sequence of instructions, each of which can be repeated. We also show how an instruction from the ARM-like Thumb-2 instruction set, `adcs`, is incorrectly categorised as an instruction for which no such sequence exists.

To sum up, the paper is structured as follows: in Section 2 we introduce and formalise the TinyARM language through a structural operational semantic. TinyARM is then used, in Section 3, as a lens through which we view the fault models of previous research in order to compare them. In Section 4 we turn our focus to the method for fault tolerance described in Moro et al., 2014 and attempt to prove it using a different approach, while Section 5 contains our example prosal for a fault tolerant version of the `adcs` instruction. Finally, we conclude the paper in Section 6.

2 The TinyARM Language

As previously mentioned, much research into fault tolerance is built up around textual descriptions of the fault model. Additionally, the specifics of the architecture on which the faults take place are often described similarly or left unspecified. As part of our contribution is to compare the fault models of various approaches, it is important that we have a common machine specification to work from. Taking inspiration from Perry et al., 2007, it is clear that having a well-defined semantic description of a language is important. Staying in the same mindset as Hansen et al., 2016, we want our language to be as closely related to a real world machine as possible while still remaining simple enough to reason about. Therefore, we do not include instructions that require special hardware support that cannot be found on regular ARM machines. In reality, this means using the same language as Hansen et al., 2016, TinyARM, with a few modifications. In this section we define the syntax and structural operational semantics of TinyARM and use these to also give a formal definition of various fault that can occur in TinyARM.

2.1 Syntax and Semantics

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

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

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

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

Given the 32-bit wide architecture of TinyARM, addresses of locations in the heap have the same size as values. Therefore we simply describe them as:

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

In keeping with Hansen et al., 2016 and the ARM architecture, we define 14 registers: 13 general purpose and the Program Counter (PC):

$$\text{GeneralRegister} = \{r_0, r_1, ..., r_{12}\}$$

$$\text{ControlRegister} = \{r_{pc}\}$$

$$\text{Register} = \text{GeneralRegister} \cup \text{ControlRegister}$$

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

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

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

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

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

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

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

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

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

Additionally, we define the $\neg$ operator for condition codes as the following:

$$\neg\chi = \begin{cases} \mathsf{NE} & \text{if } \chi = \mathsf{EQ} \\ \mathsf{EQ} & \text{if } \chi = \mathsf{NE} \\ \mathsf{CC} & \text{if } \chi = \mathsf{CS} \\ \mathsf{CS} & \text{if } \chi = \mathsf{CC} \\ \mathsf{PL} & \text{if } \chi = \mathsf{MI} \\ \mathsf{MI} & \text{if } \chi = \mathsf{PL} \\ \mathsf{VC} & \text{if } \chi = \mathsf{VS} \\ \mathsf{VS} & \text{if } \chi = \mathsf{VC} \\ \mathsf{LS} & \text{if } \chi = \mathsf{HI} \\ \mathsf{HI} & \text{if } \chi = \mathsf{LS} \\ \mathsf{LT} & \text{if } \chi = \mathsf{GE} \\ \mathsf{GE} & \text{if } \chi = \mathsf{LT} \\ \mathsf{LE} & \text{if } \chi = \mathsf{GT} \\ \mathsf{GT} & \text{if } \chi = \mathsf{LE} \\ \mathsf{NV} & \text{if } \chi = \mathsf{AL} \\ \mathsf{AL} & \text{if } \chi = \mathsf{NV} \end{cases}$$

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

We argue that the introduction of TinyARM by Hansen et al., 2016 has some inconsistencies in its definition of the arithmetic operations and their effect on **Flags**. Specifically, the + operator is used both for addition of binary values and regular integers, and it is unclear when two's complement representation of values is taken into account. For example it is stated that the overflow flag is set on addition if $v_1, v_2 > 0 \;\wedge\; (v_1 + v_2 \geq 2^{31})$. Intuitively, this is the idea that if the sum of two positive numbers is negative, an overflow must have occurred. However, checking if the values are positive with $v_1, v_2 > 0$ assumes they are mathematical values, whereas the sign of the resulting from the addition is checked as $(v_1 + v_2 \geq 2^{31})$, which assumes the result is in two's complement. The same problem is present in the other case where the overflow flag is set: $v_1, v_2 < 0 \;\wedge\; (v_1 + v_2 \geq 0)$. To avoid these problems we define our own functions for addition and subtraction and make use of the fact that our binary values allow us to extract individual bits.

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

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

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

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

$$add_{bin} : \mathbb{B}_n \times \mathbb{B}_n \times \{0,1\} \to \mathbb{B}_n$$
$$add_{bin}(v_1, v_2, c_{in})(n) = v_1(n) + v_2(n) + carry_{bin}(v_1, v_2, c_{in})(n) \mod 2$$

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

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

$$\begin{aligned}&sub_{bin} \colon \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n\\ &sub_{bin}(v_1, v_2)(n) = add_{bin}(v_1, inv_{bin}(v_2), 1)\end{aligned}$$

With these functions defined, it is now possible to give a proper definition of the $flags_{\mathsf{ADD}}$ function:

$$flags_{\mathsf{ADD}} \colon \mathbf{Val} \times \mathbf{Val} \to \mathbf{Flags}$$
$$flags_{\mathsf{ADD}}(v_1, v_2)(f_N) = add_{bin}(v_1, v_2, 0)(31)$$

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

$$flags_{\mathsf{ADD}}(v_1, v_2)(f_C) = carry_{bin}(v_1, v_2, 0)(32)$$

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

The negative flag, f_N, is set whenever the 32nd bit of the result of adding the two bit strings together is set. If a value is regarded as a two's complement signed integer, the 32nd bit being set means the value is negative. The zero flag, f_Z, is set if all bits of the result are set to zero. For the carry flag, f_C, we make use of the fact that the $carry_{bin}$ function is defined for bit strings of arbitrary length and simply set the flag to the value of the 33rd bit of the carry. If we regard the inputs and the result as two's complement signed integers, the overflow flag, f_V, can be said to be set in two cases: if we add two positive values and the result is negative or if we add two negative values and the result is positive. This closely follows the definitions found in the ARM Reference (ARM, 2005).

$$flags_{\mathsf{SUB}} \colon \mathbf{Val} \times \mathbf{Val} \to \mathbf{Flags}$$
$$flags_{\mathsf{SUB}}(v_1, v_2)(f_N) = sub_{bin}(v_1, v_2)(31)$$

$$flags_{\mathsf{SUB}}(v_1, v_2)(f_Z) = \begin{cases} 1 & \text{if } \forall j \in [0, 1, \ldots, 31] \quad add_{sub}(v_1, v_2)(j) = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$flags_{\mathsf{SUB}}(v_1, v_2)(f_C) = carry_{bin}(v_1, inv_{bin}(v_2), 1)(32)$$

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

The logic behind setting the flags when subtracting is similar to addition with a few modifications.

For the carry flag, f_C, it is worth noting that we again use the fact that subtraction is the same as negating the second argument and performing addition.

In contrast to Hansen et al., 2016 we have no need for the TinyARM variation that includes blue/green instructions. Because of this, we simply define the observable variation of the language as one entity. We name the set of all instructions in TinyARM *Instr* and define a member of this set with the following grammar:

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

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

With this we can formalise a program as a mapping of addresses to instructions and heap memory as a mapping of addresses to values:

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

Exactly as Hansen et al., 2016, we formalise the configurations used in the structural operational semantics:

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

Additionally, we adopt what we assume to be a common notational convenience when dealing with the semantics of assembly languages. With $R = \mathbf{Registers}$ and for $x \in$ Register and $n \in \mathbb{Z}$ we define

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

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

With this we can define the semantics of the language:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We further define $\Longrightarrow^n$ as a sequence of n reductions. It is worth noting that the [cmp]-rule updates the flags using $\mathit{flags}_{\texttt{SUB}}$. This is in accordance with the description of comparison found in the Arm Reference Manual, as a comparison is simply performed as a subtraction that does not store the result, but updates the control flags (ARM, 2005).

2.2 Fault Models

Having a rigorous definition of TinyARM gives us a solid foundation for modelling how a simple, but realistic, ARM-like machine works. Being able to formally describe a machine operating correctly also gives us the ability to describe what occurs when it does not operate correctly. We do so by formulating specific semantic rules for the different types of faults that can occur. In these rules we annotate the reduction relation with the type of fault that has occurred:

$$C \Longrightarrow_{\phi} C'$$

where $C, C' \in \text{Conf}$, $\phi \in \mathcal{F}$ and $\mathcal{F}$ is the fault model, i.e. the set of faults that can occur. Similar to $\Longrightarrow^{n}$, $\Longrightarrow^{n}_{\phi}$ describes a sequence of n reductions where one fault, ϕ, has occurred at some point. Additionally, we need a formal way of describing what happens to a bit string when an SEU occurs. Once again, we follow the definitions of Hansen et al., 2016 and define what it means for two bit strings, $v_1, v_2 \in \mathbb{B}_n$ to differ by exactly one bit, i.e. have a Hamming distance 1:

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

All fault models are prefixed with *f*- to differentiate them from the general semantic rules. As discussed in the example in the introduction, faults can occur both in general registers and control registers, such as the program counter. Therefore, we create two rules for SEUs in registers, one for faults in any of the general registers and one for faults in a control registers:

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

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

Again, as the example in the introduction shows, changing the single bits stored in individual flags can have a significant impact on program behaviours. Therefore we introduce the following fault:

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

Real-world architectures often make use of an instruction register, i.e. a register that stores the instruction about to be executed. If a fault occurs in this register, the program itself can be changed in a non-permanent manner, i.e. one that does not change the program itself. While our TinyARM machine does not explicitly contain an instruction register, we can model the same behaviour using the existing definitions, save for one: We require the ability to talk about the binary encoding of each instruction (their opcodes), such that we can talk about which instructions have a Hamming distance of one between them. We simply define the function *opcode*: *Instr* $\to \mathbb{B}_{32}$ to return the binary encoding of an instruction. The rule can then be defined as:

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

Generally, research often refers to memory as being protected from faults due to the simplicity of incorporating Error Correcting Codes (ECC) in memory as compared to the complexity required to protect individual parts of a CPU (Hansen et al., 2016; Reis et al., 2005; Perry et al., 2007; Moro et al., 2014). However, new research shows that certain Rowhammer exploits can flip bits in ECC protected memory (Jeong et al., 2019). Regardless of its real world usage, we define how SEUs in memory would behave. Memory encompasses both the program itself and all the data stored in the heap. Faults in these two locations have quite differing effects, so we model them as separate faults:

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

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

Finally, we define the fault that any one instruction is skipped. This could technically also occur under any combination of $[f\text{-reg}_{\text{ctrl}}]$, $[f\text{-instr}]$, $[f\text{-mem}_{\text{prog}}]$ either by jumping one instruction ahead or by changing any instruction into a `NOP` instruction. Depending on the opcodes of individual instructions it might not be possible for an SEU to do this, however we would argue that there are several ways in which an instruction can be skipped without specifically being converted into a `NOP`, especially each instruction can be executed conditionally. Therefore, we simply define the fault as follows:

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

Both $[f\text{-instr}]$, $[f\text{-skip}]$ and $[f\text{-mem}_{\text{prog}}]$ are examples of faults that could lead to the situation described in the example in the introduction, where the intended effects of a single instruction are not achieved.

3 Review

In the following section we cover a number of techniques for providing fault tolerance that has been the result of previous research. Table 1 contains a cursory overview of the examined techniques and the types of faults they protect against. This is followed by a section for each paper, which discusses the technique presented in greater detail. Note that the technique by Nicolescu and Velazco, 2003 differs from the rest in that it operates on C-code rather than some type of assembly language.

Paper	Name (if any)	Registers vulnerable	Memory vulnerable	Platform
Oh, Shirvani, and McCluskey, 2002a	CFCSS	PC	No	R4400 (MIPS)
Oh, Shirvani, and McCluskey, 2002b	EDDI	General, PC, Flag	Yes	R10000 (MIPS)
Reis et al., 2005	SWIFT	General, PC, Flag	No	Itanium 2 (IA-64)
Perry et al., 2007	TAL_{FT}	General	No	TAL_{FT}
Moro et al., 2014		PC	Program code	Thumb-2
Hansen et al., 2016	TinyARM	General, PC	No	TinyARM
Nicolescu and Velazco, 2003	C2C	General	Yes	DSP32C

Table 1: Techniques reviewed

For each paper we give a quick introduction to how the technique works in order to highlight the main contribution of each paper. Additionally, we use the rigorously defined TinyARM language to view the technique from the perspective of hardware that is closer to the real world. We do so, by exemplifying how the faults allowed in each method equate to the different types of faults defined in Section 2.2.

It is important to note that the cardinality of the fault models alone cannot be used to compare the effectiveness of each technique. This is mainly because a fault model does not actually capture any information about the fault tolerance provided by a technique. Formally comparing fault models might seem nonsensical when we assume the techniques are run in a imaginary machine, but we would argue that it is an important steppingstone towards a more formal language for comparing different software solutions for fault tolerance.

3.1 Control-Flow Checking by Software Signatures

The method presented by Oh, Shirvani, and McCluskey, 2002a, CFCSS, is focused on providing detection of errors in control-flow. They accomplish this by separating programs into basic blocks at compile-time and assigning each block a signature. The signature is calculated based on the signature of the block before it in the control-flow graph. At run-time the signature is recalculated and checked against the statically calculated one each time a control-flow transfer happens. This makes it possible to detect if an illegal transfer of control ever takes place. The authors argue that most ways of illegally transferring control are caught by this scheme, sometimes with a slight delay before detection, such as when the checking instructions in a basic block are skipped. In this case, the next legal control transfer will lead to a mismatch between signatures as a step of the calculation has been skipped. Their technique does not capture an illegal transfer of control within a basic block though, as this has no effect on the signature or its calculation. Additionally, it is possible for parts of a basic block to be skipped if control is transferred to the beginning of the next basic block and likewise conditional jumps that follow the wrong branch are not detected. Under the assumption that programs have the objective of outputting some result of their calculations to a memory mapped output device, performing the check upon entering each basic block is not strictly necessary. Rather, the checks can be deferred to basic blocks which contain a store instruction. This idea is presented here, but is not utilised. The fault model assumed is not explicitly stated in the paper, but it is reasonable to expect that faults are allowed in the PC, as this is a prime candidate for control-flow errors. As CFCSS makes use of a general purpose register to store the current run-time signature, it is safe to assume that these registers are protected from faults. Likewise, the control flags must be protected, as the comparison of compile-time and run-time signatures makes use of them. Finally, no changes can be made to individual instructions in the program, as CFCSS has no way of recognising that type of fault and it could hinder the correct detection of control-flow faults. There is mention of control-flow errors being able to originate from faults in memory or the instruction currently being executed, but it would have to SEUs specifically targeting the addresses in branching instructions. The effect of this is already captured in our modelling of faults in the program counter. Therefore, the fault model of CFCSS can simply be summed up as:

$$\mathcal{F} = \{[f\text{-reg}_{\text{ctrl}}]\}$$

3.2 Error Detection by Duplicated Instructions in Super-Scalar Processors

The method presented by Oh, Shirvani, and McCluskey, 2002b, relies on duplicating instructions in a way that many other later techniques also take advantage of. Specifically, each original instruction in a program, a Master Instruction (MI), is duplicated to produce a Shadow Instruction (SI). The SI must use a different set of registers and heap addresses than its corresponding MI, thus the set of registers and the set of memory addresses used by a program are each partitioned into two disjoint subsets. Every calculation is then performed twice, once in each set of registers. Multiple schemes for the arrangement of MIs and SIs are given, with a focus being on finding an interleaving scheme that maximises the number of illegal control transfers that can be caught while also taking advantage of the fact that filling the instruction window on a super scalar processor with independent instructions (which MIs and SIs are prime examples of) can lead to lower overhead of using the technique. The authors argue that a control-flow errors can be caught anytime either an MI or an SI is skipped, as this would leave the end results of the two chain of calculations in different states. This seems to a bit of an oversimplification, as it relies on each instruction having an effect on the values, which does not always have to be the case, rather it would depend on a given program and its input. The results are compared every time they are to be stored in memory or a branch is about to happen. If this comparison ever shows differing values a fault must have occurred, the authors argue. When saving to memory, both the master and shadow version of the value must be saved, albeit to different locations in the heap, such that they can be loaded into different registers again at a later time, without ever having been mixed. The obvious implication of the fact that data memory is being protected from faults by the technique is that it is part of the fault model. Additionally, with the focus on calculating the probabilities of SEUs converting instructions into other instructions, both program memory and the corruption of the instruction register is part of the fault model. Despite no mention of it by Oh, Shirvani, and McCluskey, 2002b, we would argue that SEUs in general registers are also discovered by the method, as these would inevitably be caught by the comparisons they make between the contents of MI registers and SI registers. For faults in control flags, the comparisons would fail, leading to discovered faults. Given their argument that control flow errors are also discovered as they would lead to a mismatch between the number of MI and SI executed, SEUs in the program counter can also be allowed. The fault model of EDDI then, is:

$$\mathcal{F} = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}], [f\text{-flag}], [f\text{-instr}], [f\text{-mem}_{\text{prog}}], [f\text{-mem}_{\text{data}}]\}$$

3.3 SWIFT: Software Implemented Fault Tolerance

The technique introduced by Reis et al., 2005 is presented as an evolution on previous work, namely CFCSS and EDDI. Specifically, the method is described as EDDI, with half the memory requirements merged with an improved version of CFCSS. The improvement to memory use comes from acknowledging that memory is often well protected through ECC, thereby removing the need to duplicate all data stored in memory. CFCSS is improved in two ways: Firstly, as originally described by Oh, Shirvani, and McCluskey, 2002a, it is not necessary to check the run-time signature each time a basic block is entered, only when entering blocks containing store instructions, as these produce the only observable changes of programs. This cuts down on the number of comparison instructions injected into programs, reducing the overhead of SWIFT. Secondly, CFCSS is only able to detect illegal branches if they are not contained in the control-flow graph of the program in question. This precludes the ability to detect faults that force control down the wrong branches of conditional branches, as control is still transferred to a legal address. This is solved by what the authors dub *enhanced control-flow checking* where the end of each block contains instructions to assert which address control is about to be transferred to, while the beginning of each block confirms that the assertion holds. While not explicitly mentioned by the authors, this also solves a problem with CFCSS where the remaining instructions of a given basic block could be skipped without being detected, if control was ever transferred to the beginning of the next legal basic block. Given that this technique is essentially a combination of EDDI and CFCSS, it makes sense that the fault model is similar to the union of those methods' fault models, with the exclusion of memory which is now assumed to be protected. The means that the fault model can be described as:

$$\mathcal{F} = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}], [f\text{-flag}], [f\text{-instr}]\}$$

3.4 Fault-tolerant Typed Assembly Language

Perry et al., 2007 presents a technique that conceptually does not differ much from EDDI, in that it relies on calculating all values twice, using two disjoint sets of registers. The difference lies in the fact that the authors of EDDI simply assume that fault tolerance is a quality that arises from this duplication of calculations, whereas Perry et al., 2007 give a formal proof that this is the case when the two chains of calculations never influence each other. In order to keep track of which calculations belong to which chain, each calculation and register is annotated with at colour, green or blue. In order to prove it, they introduce the structural operational semantics and type system for an assembly language that is designed to run on imaginary ARM-like hardware designed by them. The authors have aimed for a language that is closely related to real ARM assembly code. However, their machine requires specialised hardware and the language includes additional instructions that make use of these changes, such that they can prove the fault tolerance of the technique. There are two main differences between TAL_{FT} and a real machine: firstly, the program counter is duplicated and all instructions related to control-flow atomically compares the value in both program counters before reading it or updating both of them. We would argue that the language can be made significantly simpler without this change, and that we can simply model the same behaviour by not including $[f\text{-reg}_{\text{ctrl}}]$ in the fault model. The second change is that store instructions must atomically compare the green and blue versions of the value about to be stored and only store it if they are equal. The atomicity of the instruction is vital, since a fault could otherwise change the value after a successful comparison, but before it is stored. This is accomplished using specialised hardware and extra instructions: the green store instruction places its operands (the value to be stored and the address to store it at) in a queue and the next blue store instruction compares its operands to the ones placed in the queue. If there is a mismatch between the values from the green and blue chain of calculations, an error has occurred. Given the formal nature of the paper as a whole, it comes as no surprise that the fault model is very well defined. Despite modelling an instruction register through their semantics they do not allow SEUs to affect an instruction stored there. Couple this with the fact that the entirety of memory is protected and it is clear that no faults can change the program itself. Additionally, as control flags are not present in TAL_{FT} they are not included in the fault model. Overall, this means that translating this technique to TinyARM will leave us with a very simple fault model:

$$\mathcal{F} = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}]\}$$

3.5 Formal Verification of a Software Countermeasure against Instruction Skip Attacks

The technique presented by Moro et al., 2014 relies on duplicating instructions, but not for the purpose of separating calculations and detecting when they differ, as with previous techniques. Instead the idea is that the redundancy introduced into the program by calculating the values multiple times will simply mask any faults that skip a single instruction. The authors attempt to group instructions in the ARM-like instruction set, Thumb-2, based on their behaviour when duplicated: Instructions which are idempotent, i.e. subsequent execution have no effect beyond the first execution, are simply considered fault tolerant if repeated. An example of an idempotent instruction is `MOV`, which simply copies a value into another register. Performing this action once or twice has the same effect and if either the first, second or none of the copies of this instruction are skipped, the result is exactly the same. The other group of instructions are those that cannot simply be repeated to become fault tolerant without changing the outcome of their execution. An example could be if the `ADD` instruction shares a source and a destination register. In this case, repeated executions will keep adding something to the same register. Many instructions such as this can be rewritten using a series of idempotent instructions to accomplish the same thing, but in a manner where duplication of the individual instructions provides fault tolerance without changing the outcome of the calculation. This group of instructions is called separable. For all but a few exotic instructions, Moro et al., 2014 manage to find rules for rewriting, such that large parts of programs can be hardened against faults which skip single instructions. For some of the instructions it is indeed difficult to imagine a possible way to rewrite them without changing the outcome, such as for instructions which are used for communicating with a co-processor or those whose semantics are left up to the individual chip designer. However, one specific instruction, `adcs`, which is used to add two values and the value stored in the carry flag, is said to be impossible to create a fault tolerant version of. In Section 5 we show that it is indeed possible, both in Thumb-2 but also in TinyARM with some small additions to the language. For all idempotent and separable instructions, model checking is used to simulate all possible configurations in an imaginary 4-bit machine to show that the technique does indeed provide fault tolerance and does not change the outcome. This proof is rather lacking, as 4-bit machines are virtually non-existent, and in Section 4 we aim to formally prove some of the authors' assumption using the well-defined semantics of TinyARM. The fault model can be described as:

$$\mathcal{F} = \{[f\text{-reg}_{\text{ctrl}}], [f\text{-instr}], [f\text{-mem}_{\text{prog}}]\}$$

3.6 Formal Modelling and Analysis of Bitflips in ARM Assembly Code

As with Perry et al., 2007, the main contribution of Hansen et al., 2016 is not in the technique itself, which again involves duplicating all calculations and making sure that the two resulting chains of calculation never influence each other. Using the language presented in the paper, TinyARM, enables the possibility of statically verifying that a program is structured such that the calculations are disjoint. TinyARM is closely related to TAL_{FT}, but better models a real world ARM-like machine, by removing the need for special hardware support and a redundant program counter. Instead of modelling a fault tolerant program counter in the language, two distinct fault types are given, one for general registers and one which can influence the program counter. By removing the last one from the fault model, there is no longer any need for a redundant program counter. The need for special hardware, however, cannot be entirely removed without reducing the number of faults that can be discovered. A number of rules for emulating the atomic compare-and-store instructions, using sequences of regular instructions that does not require hardware support, is presented. The efficacy of these rewriting rules is proven using model checking, with the exception of an edge case that makes it possible for an erroneous value to be written to memory before it can be detected that it is the result of a fault. The fault model used in most of the paper is as follows:

$$\mathcal{F} = \{[f\text{-reg}_{\text{gen}}], [f\text{-flag}]\}$$

Statistical model checking is used to also give an estimate on the number of faults the same technique can discover under a more aggressive fault model, such as when the program counter and instructions are not longer protected. That is, a fault model of:

$$\mathcal{F} = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}], [f\text{-flag}], [f\text{-instr}]\}$$

3.7 Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results

As previously mentioned, the technique presented by Nicolescu and Velazco, 2003 operates differently than the other techniques reviewed, even if it does build on some of the same fundamental ideas. Instead of duplicating or transforming code at the assembly level, this technique works at a higher level of abstraction, specifically by transpiling C-code. In doing so, a number of additional variables and conditionals are introduced into the source program to create a version that is more tolerant to faults. The authors split the transformation rules into three distinct parts, each protecting against a different type of fault. In order to protect against faults that change data, all variables except *final variables* are duplicated along with any calculation done on them. Final variables, are those that are not *intermediary*, i.e. used in the calculation of other variables. When a final variable is written to, the two independent copies of the intermediary variables they are calculated from, are compared. If they differ an error has occurred. The idea of final variables closely resembles the notion of observable changes as described by Hansen et al., 2016 while the idea of duplicating variables and the calculations they are used in, is very similar to the method described by Oh, Shirvani, and McCluskey, 2002b and refined by Reis et al., 2005. The second type of transformation rule aims to provide detection of faults that causes erroneous transfer of control as a result of changes to the program counter, if a non-branching instruction is changed to a branching one or if the address of an unconditional jump is changed. This is accomplished with the use of a global boolean variable that is made incremented modulo 2 each time a basic block is entered and each time one is exited, such that it always has value 0 when a block is active and 1 otherwise. The variable is xor'd with a signature that is unique for each basic block and the result is stored. At the end of the basic block, the stored value and the signature of the basic block are checked for equality. Only if the global status variable was 0 when the block was activated and if the signature was the correct one for the recently exited basic block, will this check pass. The idea of keeping track of which basic blocks are currently active and assigning them a signature resembles the notion of signatures used in CFCSS. Finally, faults that can impact conditional and unconditional branching are guarded against using the third transformation rules. For conditional branching, this is simply achieved by performing an additional but identical comparison directly following the original one. If the result of the second comparison differs from the first, an error must have occurred. Unconditional branching takes place when procedures are called and when they return. Detecting errors in these, is simply done by writing a unique signature to a variable before any unconditional transfer of control and checking if the expected value is stored in the variable at the address that is branched to. As with the other techniques reviewed, Nicolescu and Velazco, 2003 also includes results from benchmarks in which SEUs are injected into programs to show the efficacy of their contribution. Unlike other benchmarks we have reviewed, the authors have actually performed several tests where they test their technique under different fault models. In the first trial, it is simply stated that registers in the chip are targeted, and it is unclear if this only refers to the general registers or if the program counter is also vulnerable. Given that the technique includes some protection against faults in the program counter, we choose to include it. The second and third trials introduced faults in the program memory and data memory respectively.

$$\mathcal{F}_1 = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}]\}$$

$$\mathcal{F}_2 = \{[f\text{-instr}], [f\text{-mem}_{\text{prog}}]\}$$

$$\mathcal{F}_3 = \{[f\text{-mem}_{\text{data}}]\}$$

Additionally and uniquely for this paper, is the inclusion of a practical experiment that involves subjecting the same chips to a source of radiation and observing the resulting faults in both hardened and non-hardened programs. It is difficult to ascertain what the fault model was in this experiment, but we assume that all types of faults we have described could occur in such an environment:

$$\mathcal{F}_{\text{rad}} = \{[f\text{-reg}_{\text{gen}}], [f\text{-reg}_{\text{ctrl}}]\},$$

3.8 Findings

Based on our review of a number of existing techniques for providing fault tolerance primarily through software we conclude that simply comparing techniques based on their fault models is of limited use. The review shows that most approaches can be categorised under two paradigms: one which aims to detect whenever illegal transfer of control occurs by inserting extra control instructions between basic blocks. The second attempts to avoids logical errors by performing all calculations twice and making sure the two results are equivalent. Making more meaningful comparisons than these is difficult, as there are variables other than their fault models on which the techniques differ. It is possible to argue that the introduction of a formalised fault model simplifies the problem of deciding if one technique is better than another because it catches more errors or because fewer parts of a computer are considered vulnerable. However, the amount and type of faults that can go undetected under each technique is no simpler to convey with the help of a formalised fault model. This is especially true when comparing techniques that offer tolerance in varying ways and are supposed to be used on different types of hardware.

While we found that fault models alone was not enough to compare two techniques, we argue that presenting a well defined fault model along with a novel technique is useful, as it helps communicate exactly what the method covers and what it does not. However, such a fault model builds on a language with well defined semantics, and defining slightly different languages along with each new technique is not particularly helpful either. Instead, we propose that a standardised language much like TinyARM is developed. Small enough to not encumber research and close enough to real world machines that low level details can be included and their effects studied. Such a language should come with a set of commonly used fault models and the possibility of adding small extensions if necessary. This could help reduce ambiguity when presenting novel techniques and encourage formal verification of their efficacy, while also making it easier to compare new methods. For some techniques, there would definitely be differences between the idealised formally defined language and the real world system on which it will be used, but perhaps it would be worth it, for the increase in interoperability between researchers.

It is interesting to see how all the reviewed papers include some type of benchmark in order to show how well the technique works, but it is difficult, bordering on impossible, to use these for comparing individual techniques. A large problem lies in not knowing the exact mechanics of how the faults have been introduced in the programs. In some of the benchmarks we review, two considerations have been made when simulating faults: all of the techniques reviewed make programs longer by inserting additional instructions, which means that the programs have a larger memory footprint and that they run for longer. Both increase the probability of an SEU occurring, which has been modelled in some benchmarks by introducing a number of faults based on the both the runtime and the size of the program. However, this does not capture the fact that in the real world SEUs tend to show up more often in some parts of CPUs than others. Especially memory is often much more susceptible to faults due to its large physical size compared to individual registers. This also means that caches, which make up a relatively large part of many modern CPUs, are prime candidates for faults when compared to the relatively small size of an ALU. Even then, the floating point units in ALUs contain a large amount of combinational logic, which make them more susceptible than the rest of the ALU (Swift et al., 2001). Through radiation testing of the PowerPC750, Swift et al., 2001 also found that SEUs are more likely to flip bits from 1 to 0 than the other way. This should make it clear that if one wants to properly simulate SEUs in a given machine and architecture, it requires a deep understanding of not only how the chip is designed, but also the laws of physics that basic gate design relies on. It might not be feasible to create a benchmark which accurately depicts how faults occur in the real world, but researchers in the area could work towards creating a framework that is more advanced than the methods we have seen in our review. It would also be helpful to have use a standardised test methodology, such that new research could be compared more easily. It would be difficult to design such a methodology in a way that is platform and architecture agnostic, which means that techniques that are developed with a specialised platform in mind, might not be able to utilise it.

4 Formalisation of the Technique by Moro et al.

In this section we wish to clarify how we understand the argumentation for the classification if instructions presented by Moro et al., 2014. We do so, in order to give define alternative classes, which are grounded in our own formal definitions of the TinyARM language. This allows us to use the rigorously defined semantics to give proofs that are stronger than the model checking used by the original authors.

4.1 Classes of Instructions

As mentioned in Section 3.5, Moro et al., 2014 separates the Thumb-2 instruction set into three classes. Our previous discussing only covered two of these, *idempotent* and *separable* instructions. Idempotent instructions are those that achieve fault tolerance by simple duplication, while separable instructions are the instructions for which a decomposition into a sequence of idempotent instructions is possible. Each of the individual idempotent instructions can then be duplicated to achieve fault tolerance. The third class, the *specific* instructions, contains the remaining instructions from Thumb-2. Some instructions in this class can be decomposed into a sequence of instructions, such that fault tolerance can be achieved without altering the result. This is not the case for all instructions in the class: the authors acknowledge that some instructions in this class can only achieve partial fault tolerance while others cannot be made fault tolerant at all. The following sections highlights some problems with the classification performed my Moro et al., 2014.

4.1.1 Idempotent Instructions

The class that the whole technique arguably builds on is poorly defined. For example, fault tolerance does not follow from the authors' definition of idempotent instructions as simply as presented. They define them as follows: "Idempotent instructions are the instructions that have the same effect when executed once or several times."(Moro et al., 2014) They then go on to say that fault tolerance can be achieved by repeating any idempotent instruction. In the presence of a program counter responsible for pointing at the next instruction to be executed, this argument partly breaks down with the inclusion of instructions that can change the value of the program counter. An example of this is a branching instruction, which one can argue does not change behaviour when executed twice: it simply transfers control to a different address. Conceptually, duplicating a branch instruction does indeed provide fault tolerance: the result is the same whether the first, second or none of the instructions are skipped. However, the fault tolerance is not achieved because the instruction is idempotent: only one of the instructions is ever executed, except if they both point to the address of the second branch instruction, in which case control is stuck in an infinite loop. Grouping instructions which achieve fault tolerance when duplicated is not inherently problematic, but the reasoning for putting them there needs to clear.

4.1.2 Separable Instructions

According to Moro et al., 2014 this class contains any instruction which can be rewritten to a sequence of idempotent instructions. They argue that this encompasses several types of instructions, but again the definitions are too vague to be useful. Instructions which share a source and destination register belongs in this class they argue, but this includes MOV, an instruction that clearly belongs in the first class. Additionally, it is argued that instructions which both read and write flags belong in the specifics class, yet instructions that use a RRX shift (which reads from flags) and write the flags are classified as separable, despite the obvious similarity to the instructions placed in the specific class.

4.1.3 Specific Instructions

Moro et al., 2014 argue that the instructions in this class can actually be further classified into three distinct classes. The fact that they group them together regardless, makes the class seem like an afterthought compared to the previous two classifications. Included in the specific instructions are the instructions that cannot "easily" be replaced by a list of idempotent instructions, instructions for which fault tolerant replacement sequences do not exist and those for which a replacement sequence exists under certain constraints. An example of an instruction for which an equivalent sequence of idempotent instructions cannot easily be found, is the BL instruction, which writes the address of the next instruction to the link register before jumping to a specified address. This instruction is replaced by three idempotent instructions and an extra label. From this, we conclude that "not easily replaced" refers to the addition of labels or recalculations of addresses in the replacement sequence. The authors highlight ADCS as an instruction for which a strictly fault tolerant replacement sequence cannot be found, but we would argue this is simply not true, as discussed in Section 5. Finally, there are certainly some instructions for which a fault tolerant replacement sequence cannot exist because the semantics of the instructions depend on factors in programs outside the currently running one or because they have an effect on programs other than the currently running one.

4.2 Redefined Classes of Instructions

In this section we give our own definitions of classes that instructions can be put into, hoping to remove any ambiguity and inconsistencies. Initially, we give an informal description of each class with formal ones following in the next subsections. Before doing so, we need to redefine a few terms that will allows us to be more precise when referring to instructions, as we have used this term to refer to different concepts until now. Firstly, when we refer to instructions, we are talking about fully instantiated instructions with no logical variables. For example, $\texttt{ADD}_{\texttt{AL}}\ r_0, r_1, r_2$ and $\texttt{ADD}_{\texttt{AL}}\ r_0, r_0, r_1$ are both instructions. When we wish to refer to instructions where operands and condition codes follows a specific pattern, we talk about instructions with a specific schema. In keeping with the example, the two previous instructions have two distinct schemas: $\texttt{ADD}_{\chi}\ x, y, z$ and $\texttt{ADD}_{\chi}\ x, x, y$. Note that if the variables used in a schema are distinct then so are their values, e.g. $x \neq y$ in $\texttt{ADD}_{\chi}\ x, y, z$. Finally, we refer to instructions with a specific identifier when we talk about all instructions with the same identifier regardless of operands, condition codes, and specific semantic rules: thus, $\texttt{MOV}_{\texttt{AL}}\ r_1, 1$, $\texttt{MOV}_{\texttt{NE}}\ r_0, r_1$ and $\texttt{MOV}_{\chi}\ r_0, r_0$ which are all instructions with the same identifier: MOV. Additionally, we informally define what it means for some sequence of instructions to be fault tolerant: if after execution, some sequence of instructions reaches the same state whether a single arbitrary instruction in the sequence is skipped or not, the sequence is said to be fault tolerant.

Idempotent instructions are instructions that if placed directly after one another in a program, can be executed to reach the same configuration as executing only one of them, disregarding the value of the program counter in the final configuration.

Fault tolerant instructions are instructions that can simply be duplicated in order to create a fault tolerant sequence of two instructions. All idempotent instructions are trivially members of this class, as is the branching instruction.

Separable instructions are instructions for which there exists a sequence of fault tolerant instructions such that executing the fault tolerant instructions leads to the same configuration as executing the separable instruction, disregarding that the sequence might use registers other than the ones used by the separable instruction, to achieve its goal. All fault tolerant instructions are trivially members of this class.

Non-separable instructions are instructions for which no sequence of fault tolerant instructions with the same properties as those described above exists. As we are solely focusing on classifying instruction from TinyARM, we expect this class to remain empty, but for real instruction sets such instructions most likely exist. Examples could be instructions that signal something to a co-processor, which can then act on the signal in a manner that is not described in the semantics of the language. Another possibility is that the exact semantics of an instruction is left up to the chip designer, which means we simply cannot make any guarantees about it.

In the following sections we present formal definitions of the three first classes.

4.3 Idempotent Instructions

In this section we formally define what it means for an instruction to be idempotent and prove that some instructions are idempotent when they have a specific schema.

Definition 1: An instruction, *instr*, is said to be idempotent if and only if:

$$\begin{array}{l}
\forall \langle P,H,R,F\rangle \in \text{Conf } (\\
\quad (P(R(r_{pc})) = \mathit{instr} \;\wedge\; P(R(r_{pc})+1) = \mathit{instr}) \longrightarrow (\\
\quad\quad \exists \langle P,H',R',F'\rangle, \langle P,H'',R'',F''\rangle \in \text{Conf } (\\
\quad\quad\quad \langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle \;\wedge\; \langle P,H',R',F'\rangle \Longrightarrow \langle P,H'',R'',F''\rangle \;\wedge \\
\quad\quad\quad H' = H'' \;\wedge\; F' = F'' \;\wedge\; R' =_{\setminus r_{pc}} R'' \;\wedge \\
\quad\quad\quad R'(r_{pc}) = R(r_{pc})+1 \;\wedge\; R''(r_{pc}) = R'(r_{pc})+1 \\
\quad\quad) \\
\quad) \\
)
\end{array}$$

We note that any instruction with a condition code which evaluates to false is idempotent, simply because the entire instruction is skipped regardless of how many times it is executed:

Lemma 1: Any instruction, instr_χ, can be considered idempotent in contexts where $\mathit{cond}(\chi, F) =$ false:

$$\begin{array}{l}
\forall \langle P,H,R,F\rangle \in \text{Conf } (\\
\quad (P(R(r_{pc})) = \mathit{instr}_\chi \;\wedge\; P(R(r_{pc})+1) = \mathit{instr}_\chi \;\wedge\; \mathit{cond}(\chi,F) = \text{false}) \longrightarrow (\\
\quad\quad \exists \langle P,H',R',F'\rangle, \langle P,H'',R'',F''\rangle \in \text{Conf } (\\
\quad\quad\quad \langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle \;\wedge\; \langle P,H',R',F'\rangle \Longrightarrow \langle P,H'',R'',F''\rangle \;\wedge \\
\quad\quad\quad H' = H'' \;\wedge\; F' = F'' \;\wedge\; R' =_{\setminus r_{pc}} R'' \;\wedge \\
\quad\quad\quad R'(r_{pc}) = R(r_{pc})+1 \;\wedge\; R''(r_{pc}) = R'(r_{pc})+1 \\
\quad\quad) \\
\quad) \\
)
\end{array}$$

Proof:

PROVE: Lemma 1
ASSUME: 1. $\langle P, H, R, F\rangle \in \text{Conf}$
2. $P(R(r_{pc})) = \mathit{instr}_\chi$
3. $P(R(r_{pc}+1)) = \mathit{instr}_\chi$
4. $\mathit{cond}(\chi, F) = \text{false}$

$\langle 1\rangle 1.$ $\langle P, H, R, F\rangle \Longrightarrow \langle P, H', R', F'\rangle$ where $R' = R_{r_{pc}+1}$, $H' = H$ and $F' = F$

PROOF: by assumption 1, 2 and 4, [skip] and no other reduction is possible

$\langle 1\rangle 2.$ $\langle P, H', R', F'\rangle \Longrightarrow \langle P, H'', R'', F''\rangle$ where $R'' = R'_{r_{pc}+1}$, $H'' = H'$ and $F'' = F'$

PROOF: by $\langle 1\rangle 1$, assumption 1, 3 and 4, reduction [skip] and no other reduction is possible

$\langle 1\rangle 3.$ $R'' =_{\setminus r_{pc}} R'$

PROOF: by $\langle 1\rangle 2$ and definition of $=_{\setminus r_{pc}}$

$\langle 1\rangle 4.$ Q.E.D.

PROOF: by $\langle 1\rangle 1$, $\langle 1\rangle 2$ and $\langle 1\rangle 3$

Having proved Lemma 1, we are now able to prove that any instruction with the schema $\texttt{MOV}_\chi\ x, y$ is idempotent.

Theorem 1: Instructions with the schema $\texttt{MOV}_\chi\ x, y$ are idempotent:

$$
\begin{array}{l}
\forall \langle P, H, R, F\rangle \in \text{Conf}\ (\\
\quad (P(R(r_{pc})) = \texttt{MOV}_\chi\ x, y\ \ \wedge\ \ P(R(r_{pc})+1) = \texttt{MOV}_\chi\ x, y) \longrightarrow (\\
\quad\quad \exists \langle P, H', R', F'\rangle, \langle P, H'', R'', F''\rangle \in \text{Conf}\ (\\
\quad\quad\quad \langle P, H, R, F\rangle \Longrightarrow \langle P, H', R', F'\rangle\ \ \wedge\ \ \langle P, H', R', F'\rangle \Longrightarrow \langle P, H'', R'', F''\rangle\ \ \wedge\\
\quad\quad\quad H' = H''\ \ \wedge\ \ F' = F''\ \ \wedge\ \ R' =_{\setminus r_{pc}} R''\ \ \wedge\\
\quad\quad\quad R'(r_{pc}) = R(r_{pc}) + 1\ \ \wedge\ \ R''(r_{pc}) = R'(r_{pc}) + 1\\
\quad\quad)\\
\quad)\\
)
\end{array}
$$

Proof:

PROVE: $H' = H'', F' = F'', R' =_{\setminus r_{pc}} R''$ for all possible reductions

$\langle 1\rangle 1.$ CASE: $\mathit{cond}(\chi, F) = \text{true}$

ASSUME: 1. $P(R(r_{pc})) = \texttt{MOV}_\chi\ x, y$
2. $P(R(r_{pc})+1) = \texttt{MOV}_\chi\ x, y$

$\langle 2\rangle 1.$ $\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R', F\rangle$ where $R' = R_{r_{pc}+1}[x \mapsto R(y)]$

PROOF: by [mov_reg] and no other reduction is possible

$\langle 2\rangle 2.$ $\langle P, H, R', F\rangle \Longrightarrow \langle P, H, R'', F\rangle$ where $R'' = R'_{r_{pc}+1}[x \mapsto R'(y)]$

PROOF: by [mov_reg] and no other reduction is possible

$\langle 2\rangle 3.$ $R'' =_{\setminus r_{pc}} R'$

$\langle 3\rangle 1.$ $\forall r \in \text{Register} \setminus \{r_{pc}, x\}\quad R''(r) = R'(r)$
and $R''(x) = R'(y)$

PROOF: by definition of R'' from $\langle 2\rangle 2$

$\langle 3\rangle 2.$ $R''(x) = R'(x)$

$\langle 4\rangle 1.$
$$
\begin{aligned}
R''(x) &= R'(y)\\
&= R_{r_{pc}+1}[x \mapsto R(y)](y)\\
&= R(y)
\end{aligned}
$$

PROOF: by definition of R'' and R' from $\langle 3\rangle 1$ and $\langle 2\rangle 1$

$\langle 4\rangle 2.$ $R'(x) = R_{r_{pc}+1}[x \mapsto R(y)](x)$
$= R(y)$

PROOF: by definition of R' from $\langle 2\rangle 1$

$\langle 4\rangle 3.$ Q.E.D.

PROOF: by step $\langle 4\rangle 1$ and $\langle 4\rangle 2$

$\langle 3\rangle 3.$ Q.E.D.

PROOF: by $\langle 3\rangle 1$ and $\langle 3\rangle 2$ and definition of $=_{\setminus r_{pc}}$

$\langle 2\rangle 4.$ Q.E.D.

PROOF: by $\langle 2\rangle 1$, $\langle 2\rangle 2$ and $\langle 2\rangle 3$

$\langle 1\rangle 2.$ CASE: $\mathit{cond}(\chi, F) = \text{false}$

PROOF: by case assumption and lemma 1

$\langle 1\rangle 3.$ Q.E.D.

PROOF: by $\langle 1\rangle 1$ and $\langle 1\rangle 2$

We now show that instruction with the schema $\texttt{MOV}_\chi\ x, x$, i.e. instructions with the same source and destination register, are idempotent.

Theorem 2: Instructions with the schema $\texttt{MOV}_\chi\ x, x$ are idempotent:

$$
\begin{array}{l}
\forall \langle P, H, R, F\rangle \in \text{Conf}\ (\\
\quad (P(R(r_{pc})) = \texttt{MOV}_\chi\ x, x \ \wedge\ P(R(r_{pc})+1) = \texttt{MOV}_\chi\ x, x) \longrightarrow (\\
\quad\quad \exists \langle P, H', R', F'\rangle, \langle P, H'', R'', F''\rangle \in \text{Conf}\ (\\
\quad\quad\quad \langle P, H, R, F\rangle \Longrightarrow \langle P, H', R', F'\rangle \ \wedge\ \langle P, H', R', F'\rangle \Longrightarrow \langle P, H'', R'', F''\rangle \ \wedge \\
\quad\quad\quad H' = H'' \ \wedge\ F' = F'' \ \wedge\ R' =_{\setminus r_{pc}} R'' \ \wedge \\
\quad\quad\quad R'(r_{pc}) = R(r_{pc}) + 1 \ \wedge\ R''(r_{pc}) = R'(r_{pc}) + 1 \\
\quad\quad) \\
\quad) \\
)
\end{array}
$$

Proof:

PROVE: $H' = H'', F' = F'', R' =_{\setminus r_{pc}} R''$ for all possible reductions

$\langle 1\rangle 1.$ CASE: $\mathit{cond}(\chi, F) = \text{true}$ and $x = y$

ASSUME: 1. $P(R(r_{pc})) = \texttt{MOV}_\chi\ x, x$
2. $P(R(r_{pc}) + 1) = \texttt{MOV}_\chi\ x, x$

$\langle 2\rangle 1.$ $\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R', F\rangle$ where $R' = R_{r_{pc}+1}[x \mapsto R(x)]$

PROOF: by $[\text{mov}_{\text{reg}}]$ and no other reduction is possible

$\langle 2\rangle 2.$ $\langle P, H, R', F\rangle \Longrightarrow \langle P, H, R'', F\rangle$ where $R'' = R'_{r_{pc}+1}[x \mapsto R'(x)]$

PROOF: by $[\text{mov}_{\text{reg}}]$ and no other reduction is possible

$\langle 2\rangle 3.$ $R'' =_{\setminus r_{pc}} R'$

$\langle 3\rangle 1.$ $\forall r \in \text{Register} \setminus r_{pc} \quad R''(r) = R'(r)$

PROOF: by definition of R'' from $\langle 2\rangle 2$

$\langle 3\rangle 2.$ Q.E.D.

PROOF: by $\langle 3\rangle 1$ and definition of $=_{\setminus r_{pc}}$

⟨2⟩4. Q.E.D.

PROOF: by ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3

⟨1⟩2. CASE: $cond(\chi, F) = \text{false}$

PROOF: by case assumption and lemma 1

⟨1⟩3. Q.E.D.

PROOF: by ⟨1⟩1 and ⟨1⟩2

Additionally, we conjecture that several more instructions are idempotent as well. We group them based on similarities between their semantics: Instructions with the following schemas behave very similar to the two schemas in theorems 1 and 2 and are therefore idempotent:

$\texttt{MOV}_\chi\ x, v$
$\texttt{LDR}_\chi\ x, a$
$\texttt{LDR}_\chi\ x, y$
$\texttt{STR}_\chi\ x, a$
$\texttt{STR}_\chi\ x, y$
$\texttt{STR}_\chi\ x, x$

We would argue that instructions for addition and subtraction share an important characteristic with the previous instructions: as long as the source and destination registers are distinct, the semantics simply describe the writing some value that is not dependent on the program counter into a specific register. Therefore instructions with the following schemas are idempotent:

$\texttt{ADD}_\chi\ x, y, z$
$\texttt{ADD}_\chi\ x, y, y$
$\texttt{SUB}_\chi\ x, y, z$
$\texttt{SUB}_\chi\ x, y, y$

The versions of these instructions that also set control flags based on the outcome, behave exactly the same way as long as they are not conditionally executed. Instructions with the identifier `CMP` have semantics similar to the those with the identifier `SUBS`. Therefore instructions with the following schemas are also idempotent:

$\texttt{ADDS}_\chi\ x, y, z$, where $\chi = \texttt{AL}$
$\texttt{ADDS}_\chi\ x, y, y$, where $\chi = \texttt{AL}$
$\texttt{SUBS}_\chi\ x, y, z$, where $\chi = \texttt{AL}$
$\texttt{SUBS}_\chi\ x, y, y$, where $\chi = \texttt{AL}$
$\texttt{CMP}_\chi\ x, y$, where $\chi = \texttt{AL}$
$\texttt{CMP}_\chi\ x, x$, where $\chi = \texttt{AL}$

Finally, instructions with the schema $\texttt{NOP}_\chi$ are also idempotent following the same reasoning used in Lemma 1, as they have semantics identical to instructions that are not executed because their condition is false.

4.4 Fault Tolerant Instructions

In this section we formally define what it means for an instruction to be fault tolerant and prove all idempotent instructions are fault tolerant. As previously mentioned, if a fault tolerant instruction is duplicated, the same configuration (disregarding the program counter) is reached whether the first, second or none of the instructions is skipped. We argue that branching instructions behave similarly to idempotent instructions when it comes to fault tolerance, without following our definition of idempotence: if a branching instruction is executed twice, it leads to the same configuration (disregarding the program counter) as executing it once, but since the first execution manipulates the program counter, the second branching instruction is not executed. However, if the first instruction is skipped, then the second one is executed and the same final configuration is reached. We therefore define fault tolerant instructions in a way that allows for instructions that change the program counter.

Definition 2: An instruction, *instr*, is said to be fault tolerant if and only if:

$$
\begin{array}{l}
\forall \langle P, H, R, F\rangle \in \text{Conf } (\\
\quad P(R(r_{pc})) = \mathit{instr} \ \wedge \ P(R(r_{pc})+1) = \mathit{instr} \longrightarrow (\\
\quad\quad \exists \langle P, H', R', F'\rangle, \langle P, H'', R'', F''\rangle,\\
\quad\quad \langle P, H'_1, R'_1, F'_1\rangle, \langle P, H''_1, R''_, F''_1\rangle,\\
\quad\quad \langle P, H''_2, R''_2, F''_2\rangle \in \text{Conf } (\\
\quad\quad\quad \langle P, H, R, F\rangle \Longrightarrow \langle P, H', R', F'\rangle \ \wedge\\
\quad\quad\quad \langle P, H, R, F\rangle \Longrightarrow_{f\text{-skip}} \langle P, H'_1, R'_1, F'_1\rangle \ \wedge\\
\quad\quad\quad \langle P, H'_1, R'_1, F'_1\rangle \Longrightarrow \langle P, H''_1, R''_1, F''_1\rangle \ \wedge\\
\quad\quad\quad (R'(r_{pc}) = R(r_{pc}) + 1 \longrightarrow (\\
\quad\quad\quad\quad \langle P, H', R', F'\rangle \Longrightarrow \langle P, H'', R'', F''\rangle \ \wedge\\
\quad\quad\quad\quad \langle P, H', R', F'\rangle \Longrightarrow_{f\text{-skip}} \langle P, H''_2, R''_2, F''_2\rangle \ \wedge\\
\quad\quad\quad\quad H'' = H''_1 = H''_2 \ \wedge\\
\quad\quad\quad\quad R'' = R''_1 = R''_2 \ \wedge\\
\quad\quad\quad\quad F'' = F''_1 = F''_2\\
\quad\quad\quad)) \ \wedge\\
\quad\quad\quad (R'(r_{pc}) \neq R(r_{pc}) + 1 \longrightarrow (\\
\quad\quad\quad\quad H' = H''_1 \ \wedge\\
\quad\quad\quad\quad R' = R''_1 \ \wedge\\
\quad\quad\quad\quad F' = F''_1\\
\quad\quad\quad))\\
\quad\quad)\\
\quad)\\
)
\end{array}
$$

Given this formal definition of what it means for an instruction to be fault tolerant, we show that all idempotent instructions are fault tolerant instructions:

Theorem 3: All idempotent instructions are fault tolerant instructions, i.e.

$$
\begin{array}{l}
\forall \langle P,H,R,F\rangle \in \mathrm{Conf}\ (\\
\quad (P(R(r_{pc})) = \mathit{instr} \ \wedge\ P(R(r_{pc})+1) = \mathit{instr}) \longrightarrow (\\
\quad\quad \exists \langle P,H',R',F'\rangle, \langle P,H'',R'',F''\rangle \in \mathrm{Conf}\ (\\
\quad\quad\quad \langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle \ \wedge\ \langle P,H',R',F'\rangle \Longrightarrow \langle P,H'',R'',F''\rangle \ \wedge \\
\quad\quad\quad H' = H'' \ \wedge\ F' = F'' \ \wedge\ R' =_{\setminus r_{pc}} R'' \ \wedge \\
\quad\quad\quad R'(r_{pc}) = R(r_{pc})+1 \ \wedge\ R''(r_{pc}) = R'(r_{pc})+1 \\
\quad\quad) \\
\quad) \\
) \\
\longrightarrow \\
\forall \langle P,H,R,F\rangle \in \mathrm{Conf}\ (\\
\quad P(R(r_{pc})) = \mathit{instr} \ \wedge\ P(R(r_{pc})+1) = \mathit{instr} \longrightarrow (\\
\quad\quad \exists \langle P,H',R',F'\rangle, \langle P,H'',R'',F''\rangle, \\
\quad\quad \langle P,H'_1,R'_1,F'_1\rangle, \langle P,H''_1,R''F''_1\rangle, \\
\quad\quad \langle P,H''_2,R''_2,F''_2\rangle \in \mathrm{Conf}\ (\\
\quad\quad\quad \langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle \ \wedge \\
\quad\quad\quad \langle P,H,R,F\rangle \Longrightarrow_{f\text{-skip}} \langle P,H'_1,R'_1,F'_1\rangle \ \wedge \\
\quad\quad\quad \langle P,H'_1,R'_1,F'_1\rangle \Longrightarrow \langle P,H''_1,R''_1,F''_1\rangle \ \wedge \\
\quad\quad\quad (R'(r_{pc}) = R(r_{pc})+1 \longrightarrow (\\
\quad\quad\quad\quad \langle P,H',R',F'\rangle \Longrightarrow \langle P,H'',R'',F''\rangle \ \wedge \\
\quad\quad\quad\quad \langle P,H',R',F'\rangle \Longrightarrow_{f\text{-skip}} \langle P,H''_2,R''_2,F''_2\rangle \ \wedge \\
\quad\quad\quad\quad H'' = H''_1 = H''_2 \ \wedge \\
\quad\quad\quad\quad R'' = R''_1 = R''_2 \ \wedge \\
\quad\quad\quad\quad F'' = F''_1 = F''_2 \\
\quad\quad\quad)) \ \wedge \\
\quad\quad\quad (R'(r_{pc}) \neq R(r_{pc})+1 \longrightarrow (\\
\quad\quad\quad\quad H' = H''_1 \ \wedge \\
\quad\quad\quad\quad R' = R''_1 \ \wedge \\
\quad\quad\quad\quad F' = F''_1 \\
\quad\quad\quad)) \\
\quad\quad) \\
\quad) \\
)
\end{array}
$$

Proof:

PROVE: Theorem 3 i.e. that *instr* is fault tolerant if *instr* is idempotent

ASSUME:
1. $P(R(r_{pc})) = \mathit{instr} \ \wedge\ P(R(r_{pc})+1) = \mathit{instr}$
2. $\langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle \ \wedge\ \langle P,H',R',F'\rangle \Longrightarrow \langle P,H'',R'',F''\rangle$
3. $H' = H'' \ \wedge\ F' = F'' \ \wedge\ R' =_{\setminus r_{pc}} R'' \ \wedge\ R'(r_{pc}) = R(r_{pc})+1 \ \wedge\ R''(r_{pc}) = R'(r_{pc})+1$

⟨1⟩1. $\langle P,H,R,F\rangle \Longrightarrow \langle P,H',R',F'\rangle$

PROOF: by assumption 2

⟨1⟩2. $\langle P,H,R,F\rangle \Longrightarrow_{f\text{-skip}} \langle P,H'_1,R'_1,F'_1\rangle \ \wedge\ H'_1 = H \ \wedge\ F'_1 = F \ \wedge\ R'_1 =_{\setminus r_{pc}} R \ \wedge\ R'_1(r_{pc}) = R(r_{pc})+1$

PROOF: by [f-skip]

⟨1⟩3. $\langle P,H'_1,R'_1,F'_1\rangle \Longrightarrow \langle P,H''_1,R''_1,F''_1\rangle \ \wedge\ H''_1 = H' \ \wedge\ F''_1 = F' \ \wedge\ R''_1 =_{\setminus r_{pc}} R' \ \wedge\ R''_1(r_{pc}) =$

$R'(r_{pc}) + 1$

Proof: by assumption 1 and 3

⟨1⟩4. $(R'(r_{pc}) = R(r_{pc}) + 1)$

Proof: by assumption 3

⟨1⟩5. $(R'(r_{pc}) = R(r_{pc}) + 1) \longrightarrow \langle P, H', R', F' \rangle \Longrightarrow \langle P, H'', R'', F'' \rangle$

Proof: by step ⟨1⟩4 and assumption 2

⟨1⟩6. $(R'(r_{pc}) = R(r_{pc}) + 1) \longrightarrow ((\langle P, H', R', F' \rangle \Longrightarrow_{f\text{-skip}} \langle P, H''_2, R''_2, F''_2 \rangle \ \wedge \ H''_2 = H' \ \wedge \ F''_2 = F' \ \wedge \ R''_2 =_{\backslash r_{pc}} R' \ \wedge \ R''_2(r_{pc}) = R'(r_{pc}) + 1)$

Proof: by step ⟨1⟩4 and [f-skip]

⟨1⟩7. $(R'(r_{pc}) = R(r_{pc}) + 1) \longrightarrow (H'' = H''_1 = H''_2 \ \wedge \ R'' = R''_1 = R''_2 \ \wedge \ F'' = F''_1 = F''_2)$

⟨2⟩1. $(R'(r_{pc}) = R(r_{pc}) + 1)$

Proof: by step ⟨1⟩4

⟨2⟩2. $H'' = H''_1 \ \wedge \ R'' = R''_1 \ \wedge \ F'' = F''_1$

⟨3⟩1. $H'' = H' \ \wedge \ F'' = F' \ \wedge \ R'' =_{\backslash r_{pc}} R' \ \wedge \ R''(r_{pc}) = R'(r_{pc}) + 1$

Proof: by assumption 3

⟨3⟩2. $H' = H''_1 \ \wedge \ F' = F''_1 \ \wedge \ R' =_{\backslash r_{pc}} R''_1 \ \wedge \ R'(r_{pc}) + 1 = R''_1(r_{pc})$

Proof: by ⟨1⟩3

⟨3⟩3. Q.E.D.

Proof: by ⟨3⟩1 and ⟨3⟩2

⟨2⟩3. $H'' = H''_2 \ \wedge \ R'' = R''_2 \ \wedge \ F'' = F''_2$

⟨3⟩1. $H'' = H' \ \wedge \ F'' = F' \ \wedge \ R'' =_{\backslash r_{pc}} R' \ \wedge \ R''(r_{pc}) = R'(r_{pc}) + 1$

Proof: by assumption 3

⟨3⟩2. $H' = H''_2 \ \wedge \ F' = F''_2 \ \wedge \ R' =_{\backslash r_{pc}} R''_2 \ \wedge \ R'(r_{pc}) + 1 = R''_2(r_{pc})$

Proof: by ⟨1⟩6

⟨3⟩3. Q.E.D.

Proof: by ⟨3⟩1 and ⟨3⟩2

⟨2⟩4. Q.E.D.

Proof: by ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3

⟨1⟩8. $(R'(r_{pc}) \neq R(r_{pc}) + 1) \longrightarrow (H' = H''_1 \ \wedge \ R' = R''_1 \ \wedge \ F' = F''_1)$

Proof: $(R'(r_{pc}) \neq R(r_{pc}) + 1)$ is false by step ⟨1⟩4

⟨1⟩9. Q.E.D.

Proof: by ⟨1⟩1, ⟨1⟩4, ⟨1⟩5, ⟨1⟩2, ⟨1⟩3, ⟨1⟩6, ⟨1⟩7 and ⟨1⟩8

We believe that all instructions with the identifier `B` are also fault tolerant instructions, due to their similarity to idempotent instructions:

Conjecture 1: Instructions with the schemas $\mathtt{B}_\chi\ a$ and $\mathtt{B}_\chi\ y$ are fault tolerant.

Furthermore, we argue that it follows from Definition 2 that duplicating a fault tolerant instruction in any given program, creates a fault tolerant sequence of instructions, i.e a sequence of instructions that when executed reaches the same final configuration whether a single arbitrary instruction in the sequence is skipped or not:

Conjecture 2: Duplicating a fault tolerant instruction creates a fault tolerant sequence of the two instructions.

4.5 Separable Instructions

In this section we use our previous findings to formally define what it means for an instruction to be separable. The reasoning behind separable instructions is that they can be rewritten as a sequence of one or more fault tolerant instructions that reach the same final configuration as the separable instruction when executed (disregarding the fact that the rewritten sequence might use otherwise dead registers). Each of these fault tolerant signatures can then be duplicated to achieve a fault tolerant sequence.

Definition 3: An instruction, *instr*, is said to be separable into a sequence of fault tolerant instructions:

$$S = \begin{bmatrix} \mathit{instr}_0 \\ \mathit{instr}_1 \\ \dots \\ \mathit{instr}_n \end{bmatrix}$$

if and only if:

$$\begin{array}{l}
\forall \langle P, H, R, F\rangle \in \text{Conf}\ \forall P_s \in \textbf{Program}\ \ (\\
\quad P(R(r_{pc})) = \mathit{instr}\ \ \wedge\ \ \forall i \in [1..n]\ \ (P(R(r_{pc}) + i) = \mathtt{NOP}_{\mathtt{AL}}\ \ \wedge \\
\quad \forall i \in [0..n]\ \ (P_s(R(r_{pc}) + i) = \mathit{instr}_i\ \ \wedge\ \ \mathit{FaultTolerant}(\mathit{instr}_i))\ \ \wedge \\
\quad \forall i \notin [0..n]\ \ (P_s(R(r_{pc}) + i) = P(R(r_{pc}) + i)) \longrightarrow (\\
\qquad \langle P, H, R, F\rangle \Longrightarrow^n \langle P, H', R', F'\rangle \longrightarrow (\\
\qquad\quad \langle P_s, H, R, F\rangle \Longrightarrow^n \langle P_s, H'', R'', F''\rangle\ \ \wedge \\
\qquad\quad H'' = H'\ \ \wedge\ \ F'' = F'\ \ \wedge\ \ \forall r \in \mathit{RegistersUsed}(\mathit{instr})\ \ R''(r) = R'(r) \\
\qquad) \\
\quad) \\
)
\end{array}$$

where *FaultTolerant*(*instr*) is a logical proposition which is true if and only if the instruction *instr* is in the set of fault tolerant instructions and *RegistersUsed* is a function, which for a given instruction returns the members of GeneralRegister that the instruction makes use of, along with r_{pc}.

Conjecture 3: Instructions with the schema $\mathtt{ADD}_\chi\ x, x, y$ are separable into sequences of fault tolerant instructions:

$$S = \begin{bmatrix} \mathtt{ADD}_\chi\ z, x, y \\ \mathtt{MOV}_\chi\ x, z \end{bmatrix}$$

We conjecture that all other instructions with the identifier `ADD` that share a source and destination register behave similarly, and that the same is true for `SUB`. Therefore instructions with the following schemas are separable:

$\texttt{ADD}_\chi\ x, y, x$
$\texttt{ADD}_\chi\ x, x, x$
$\texttt{SUB}_\chi\ x, x, y$
$\texttt{SUB}_\chi\ x, y, x$
$\texttt{SUB}_\chi\ x, x, x$
$\texttt{LDR}_\chi\ x, x$

Conjecture 4: Instructions with the schema $\texttt{ADDS}_\chi\ x, y, z$, where $\chi \neq \texttt{AL}$ are separable into sequences of fault tolerant instructions:

$$S = \begin{bmatrix} \texttt{B}_{\neg\chi}\ a \\ \texttt{ADDS}_{\texttt{AL}}\ x, y, z \end{bmatrix} \text{ where } a \text{ is the address immediately following the last instruction.}$$

We further conjecture that all other instructions that read flags (through their condition codes) and write flags are separable in a similar fashion:

$\texttt{ADDS}_\chi\ x, y, y$ Where $\chi \neq \texttt{AL}$
$\texttt{SUBS}_\chi\ x, y, z$ Where $\chi \neq \texttt{AL}$
$\texttt{SUBS}_\chi\ x, y, y$ Where $\chi \neq \texttt{AL}$
$\texttt{CMP}_\chi\ x, y$ Where $\chi \neq \texttt{AL}$
$\texttt{CMP}_\chi\ x, x$ Where $\chi \neq \texttt{AL}$

Finally, instructions which have both of these properties are also separable.

Conjecture 5: Instructions with the schema $\texttt{ADDS}_\chi\ x, x, y$ are separable into sequences of fault tolerant instructions:

$$S = \begin{bmatrix} \texttt{B}_{\neg\chi}\ a \\ \texttt{ADDS}_{\texttt{AL}}\ z, x, y \\ \texttt{MOV}_{\texttt{AL}}\ x, z \end{bmatrix} \text{ where } a \text{ is the address immediately following the last instruction.}$$

We argue that the same is true for instructions with the following schemas:

$\texttt{ADDS}_\chi\ x, x, y$
$\texttt{ADDS}_\chi\ x, y, x$
$\texttt{ADDS}_\chi\ x, x, x$
$\texttt{SUBS}_\chi\ x, x, y$
$\texttt{SUBS}_\chi\ x, y, x$
$\texttt{SUBS}_\chi\ x, x, x$

Our formalisation of the technique presented by Moro et al., 2014 is not complete: we have only proven that some instructions are idempotent and that all idempotent instructions are fault tolerant. Nevertheless, we have done the ground work for proving fault tolerance or separability of the remaining instructions in TinyARM and for proving that an entire program can be made fault tolerant using the technique. This shows that using a formally defined language it is possible to construct stronger arguments than those achieved by model checking. Especially, since the model checking quickly becomes infeasible when attempting to model machines with realistic sized registers. In stark contrast to this, the proofs we have outlined are trivially expandable to any bit width.

Additionally, through the work of redefining the classes of instructions, we were forced to question the authors' original classification, which meant that we questioned their conclusion that the ARM instruction `ADCS` could not be separated in a way that always gave the same result if a single instruction was skipped. In the next section we describe how this can be achieved.

5 Separability of `ADCS`

In this section we show that it is indeed possible to rewrite the `ADCS` instruction to be fault tolerant in the same manner as Moro et al., 2014 does for other instructions, i.e. simply by achieving the same effect using instructions which can be duplicated to achieve fault tolerance. We show this using both the Thumb-2 instruction set as per the original paper, but also show that it can be done using TinyARM and therefore take advantage of the definitions and proofs shown in previous sections.

The `ADCS` instruction is used to add three values: the values in the two operand registers and the 1 or 0 stored in the carry flag. The idea behind our rewriting rule is that `ADCS r0, r1, r2` is simply equivalent to `ADDS r0, r1, r2` if the carry flag is not set. If the carry flag is set however, the two instructions `ADDS r3, r1, r2` and `ADDS r0, r3, 1` performs the same addition, but the carry and overflow flags will then be set by the final addition. In the original instruction these two flags would be set to what corresponds to a combination of the two additions. This can simply by achieved by saving the resulting flags in registers and writing the logical disjunction of the two into the flags. Deciding on either of the two possibilities is simply achieved by branching based on the carry flag. As all the instructions used for this have fault tolerant signatures, each one can simply be repeated to achieve fault tolerance. This is shown in Thumb-2 assembly code in Listing 1

```
    BCS carrySet          ; branch if carry flag is set
    BCS carrySet
    ADDS r1 r2 r3         ; no carry, adcs is equivalent to adds
    ADDS r1 r2 r3
    B end                 ; skip to the end
    B end
carrySet:
    ADDS r4 r2 r3         ; addition of the two values
    ADDS r4 r2 r3
    MRS r5 apsr           ; save flags resulting from first addition
    MRS r5 apsr
    ADDS r1 r4 1          ; addition of the 1 in carry
    ADDS r1 r4 1
    MRS r6 apsr           ; save flags resulting from second addition
    MRS r6 apsr
    AND r7 r5 0x30000000 ; mask other flags from first addition
    AND r7 r5 0x30000000
    OR r8 r6 r7           ; combine flags from first and second addition
    OR r8 r6 r7
    MSR apsr r8           ; set flags to combined flags
    MSR apsr r8
end:
```

Listing 1: The `ADCS r1, r2, r3` instruction, rewritten as a sequence of idempotent instructions in Thumb-2

It can certainly be argued that the rewritten version of ADCS requires a significant number of extra instructions and registers. In fact, the trivial version of the rewriting requires an additional five registers, but it is possible to reduce this number to two, if both the AND and OR instructions are idempotent when they share a source and destination register, which we would argue they are.

We also show how the same operation can be created (in a fault tolerant way) in TinyARM. This allows us to actually make use of our findings from Section 4. We do need to extend our definition of TinyARM slightly though: it is possible to conditionally load a 1 or 0 into a given register, which means we can copy the value of a flag into a register, but writing a specific value to only a single flag is a bit more complicated. Therefore we introduce the instruction MRF for doing setting a given flag to the LSB of the value stored in a register. For good measure we also include the instruction MFR for copying the value of a flag into a register:

$$\begin{array}{rcll} \mathit{instr}_{\texttt{ADCS}} & ::= & \ldots & \text{(instructions from } \mathit{Instr}\text{)} \\ & | & \texttt{MFR}_\chi\ x, f & \text{store value in } f \text{ in } x \\ & | & \texttt{MRF}_\chi\ x, f & \text{store LSB of value in } x \text{ in } f \end{array}$$

where $\chi \in$ ConditionCode, $x \in$ GeneralRegister and $f \in \textbf{Flag}$.

Additionally, the semantics of the ADCS extension of TinyARM are defined as follows:

$$[\text{mfr}] \quad \frac{P(R(r_{pc})) = \texttt{MFR}_\chi\ x, f \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto F(f)], F\rangle}$$

$$[\text{mrf}] \quad \frac{P(R(r_{pc})) = \texttt{MRF}_\chi\ x, f \quad \mathit{cond}(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, F[f \mapsto R(x)(0)]\rangle}$$

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

The operation we use the AND and OR instructions for in the Thumb-2 example can also be accomplished using conditional execution, but we deemed that introducing an additional level of conditional execution using branching would muddle the details. For TinyARM, we avoid defining new instructions for logical conjunction and disjunction, and instead make use of the fact that condition codes allow us to perform conditional execution without branching:

```
  B_CS carrySet ; branch if carry flag is set
  B_CS carrySet
  ADDS r1 r2 r3 ; no carry, adcs is equivalent to adds
  ADDS r1 r2 r3
  B end         ; skip to the end
  B end
carrySet:
  ADDS r4 r2 r3 ; addition of the two values
  ADDS r4 r2 r3
  MFR r5 Fc     ; save carry flag resulting from first ADDS
  MFR r5 Fc
  MFR r6 Fv     ; save overflow flag resulting from first ADDS
  MFR r6 Fv
  ADDS r1 r4 1  ; addition of the 1 in carry
  ADDS r1 r4 1
  MRF_CC r5 Fc  ; set carry flag to result of first ADDS, if not set
  MRF_CC r5 Fc
  MRF_VC r6 Fv  ; set overflow flag to result of first ADDS, if not set
  MRF_VC r6 Fv
end:
```

Listing 2: The `ADCS r1, r2, r3` instruction, rewritten as a sequence of idempotent instructions in TinyARM

6 Conclusion

In this paper we have redefined the language TinyARM which attempts to formalise a simple ARM-like machine. The language was originally defined by Hansen et al., 2016, but we have extended some parts of it and removed a few ambiguities. We have used the language as a foundation for formalising different types of faults and how they affect a system. These fault models are used in our review of a handful of different techniques for providing tolerance against SEUs. The fault models are used to compare the techniques, but we find that it is difficult to compare them simply based on the fault models defined by us, both because the techniques are designed for systems different than TinyARM and because techniques differ in more ways than simply which SEUs they protect against. We come to the conclusion that if authors of novel techniques used a common framework for specifying their systems, fault models and perhaps even benchmarks, and proof techniques it would be considerably easier to compare techniques and prove their efficacy. The development of such a framework is left open for future work, but we attempt to show that it is indeed possible to use a well defined language such as TinyARM to prove the correctness of a technique for which it was not designed. We do so by using structured formal proofs to show that some instructions do indeed posses properties that are asserted by Moro et al., 2014. There are still central conjectures that need proofs and definitions that can be formalised further, especially concerning the fault tolerance of separable instructions. We leave this for future work. Finally, we argue that the `ADCS` instruction from the Thumb-2 instruction set is wrongfully categorised by Moro et al., 2014.

References

Moro, N. et al. (Sept. 2014). “Formal verification of a software countermeasure against instruction skip attacks”. In: *Journal of Cryptographic Engineering* 4.3, pp. 145–156. ISSN: 2190-8516. DOI: `10.1007/s13389-014-0077-7`. URL: `https://doi.org/10.1007/s13389-014-0077-7`.

Tront, J. G., J. R. Armstrong, and J. V. Oak (Dec. 1985). “Software Techniques for Detecting Single-Event Upsets in Satellite Computers”. In: *IEEE Transactions on Nuclear Science* 32.6, pp. 4225–4228. ISSN: 1558-1578. DOI: `10.1109/TNS.1985.4334099`.

Shivakumar, P. et al. (June 2002). “Modeling the effect of technology trends on the soft error rate of combinational logic”. In: *Proceedings International Conference on Dependable Systems and Networks*, pp. 389–398. DOI: `10.1109/DSN.2002.1028924`.

Borkar, S. (Nov. 2005). “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation”. In: *IEEE Micro* 25.6, pp. 10–16. ISSN: 1937-4143. DOI: `10.1109/MM.2005.110`.

Perry, Frances et al. (June 2007). “Fault-tolerant Typed Assembly Language”. In: *SIGPLAN Not.* 42.6, pp. 42–53. ISSN: 0362-1340. DOI: `10.1145/1273442.1250741`.

Hansen, René Rydhof et al. (Oct. 2016). “Formal modelling and analysis of Bitflips in ARM assembly code”. In: *Information Systems Frontiers* 18.5, pp. 909–925.

ARM (2005). *ARMv5 Architecture Reference Manual.* 110 Fulbourn Road Cambridge, England CB1 9NJ: ARM limited.

Reis, G. A. et al. (Mar. 2005). “SWIFT: software implemented fault tolerance”. In: *International Symposium on Code Generation and Optimization*, pp. 243–254. DOI: `10.1109/CGO.2005.34`.

Jeong, D. R. et al. (May 2019). “Razzer: Finding Kernel Race Bugs through Fuzzing”. In: *2019 2019 IEEE Symposium on Security and Privacy (SP).* Los Alamitos, CA, USA: IEEE Computer Society. DOI: `10.1109/SP.2019.00017`. URL: `https://doi.ieeecomputersociety.org/10.1109/SP.2019.00017`.

Nicolescu, B. and R. Velazco (Mar. 2003). “Detecting soft errors by a purely software approach: method, tools and experimental results”. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*, 57–62 suppl. DOI: `10.1109/DATE.2003.1253806`.

Oh, Nahmsuk, Philip P. Shirvani, and Edward J. McCluskey (Mar. 2002a). “Control-flow checking by software signatures”. In: *IEEE Transactions on Reliability* 51.1, pp. 111–122. DOI: `10.1109/24.994926`.

— (Mar. 2002b). “Error detection by duplicated instructions in super-scalar processors”. In: *IEEE Transactions on Reliability* 51.1, pp. 63–75. DOI: `10.1109/24.994913`.

Swift, G.M et al. (Dec. 2001). “Single-event upset in the PowerPC750 microprocessor”. In: *IEEE Transactions on Nuclear Science* 48.6, pp. 1822, 1827. ISSN: 0018-9499.