Formal Methods for Modelling and Analysis of Single-Event Upsets

René Rydhof Hansen, Kim Guldstrand Larsen, Mads Chr. Olesen and Erik Ramsgaard Wognsen
Department of Computer Science, Aalborg University, Denmark
{rrh,kgl,mchro,erw}@cs.aau.dk

***Abstract*—When a high-energy particle such as a proton strikes a CPU, the impact may result in the corruption of a data register on the CPU. Such a *single-event upset* (SEU), in which a random bit is flipped in the content of a data register, can lead to critical errors in the execution of a program. This is particularly problematic for security- or safety-critical systems where such errors may have grave consequences. In this paper we develop a formal semantic framework for easy formal modelling of a large variety of SEUs in a core assembly language capturing the essential features of the ARM assembly language.**

We use this framework to formally prove the soundness of a static analysis enforcing so-called blue/green separation in a given program. Blue/green separation is a replication based technique for making a program fault-tolerant with respect to data-flow SEUs; however, full coverage requires special hardware support. We further use our semantic framework for deriving program fragments, so-called *gadgets*, for partial blue/green separation without special hardware. Finally, we illustrate how to apply *statistical model checking* in our framework to model and *quantify* faults that go well beyond data-flow SEUs and can provide statistics on the level of fault-tolerance of a program. We use this to provide evidence that our suggested program modifications significantly decrease the probability of such errors going undetected.

I. Introduction

On May 24, 2013 the AAUSAT3 satellite[1], a student-driven cubesat project, experienced an apparent malfunction leaving the ground station unable to communicate with the on-board electronic power supply (EPS)[2]. While not an immediately critical failure, it had the potential over time to turn the satellite into so much space junk. After initial analysis the cause of the malfunction was attributed to a *bitflip* in the memory module of the communication protocol used between the EPS and the communication module. Eventually the situation was resolved by "exploiting" a lack of input validation and rebooting the EPS.

Failure due to bitflips in memory, or *single-event upsets* as they are also known, are not unique to cheap, student-driven projects. Also more elaborate projects operated by highly professional organisations with large budgets may fall victim to the unpredictable nature of bitflips including, e.g., NASA's Cassini Spacecraft[3] and SpaceX's Falcon 9[4]. Also systems closer to earth are vulnerable to bitflips [1]. In fact, modern processor design, with an emphasis on higher clock rates at lower voltages, combined with an increasing number of transistors in still smaller spaces contribute to an expected increase in fault rates of around 8% per processor generation [2].

In this paper we continue along the lines of the pioneering work of Perry et al. [3] and show how language-based formal methods for software verification, here program analysis and model checking, can be used to reason about, improve, and even guarantee that a program is able to detect a bitflip. The focus on formal methods enables us to formally prove the soundness of our proposed solutions as well as provide a framework for exploring the use of statistical model checking [4] both to quantify the effectiveness of proposed solutions and also to reason about more complicated fault models.

In particular we formalise the semantics of a fragment of the ARM assembly language [5] in two variants: one with only the standard instructions and one that includes special instructions for explicit support of fault tolerance through the so-called "blue/green" encoding (more details below). Based on the latter language variant, a program analysis to ensure that the blue/green instructions are used in the correct way is developed and formally proved correct (Section VI). Since few real hardware platforms provide the instructions needed for the mentioned blue/green encoding, we develop a number of so-called *gadgets* that are small code snippets intended to provide the same functionality as the special instructions, but entirely in software (Section VII). In contrast to much of the existing work on software fault tolerance, our formalisation allows us to formally prove the efficacy of these gadgets, using model checking for the actual proof.

While the general blue/green approach and our gadgets are best suited to guarantee protection against bitflips in *data* registers, we show in the final part of the paper (Section VIII) how advanced fault models that are hard to reason about, including bitflips in the program counter and program code, can be examined by applying statistical model checking (SMC) to our formalisation. The use of SMC not only allows us to explore more complex fault scenarios, it also provides a practically useful tool, that scales well to real-life problems, for providing statistical guarantees/measures of how well (or bad) proposed fault tolerance solutions work.

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

- An elementary formalisation of semantics and fault models for a low-level target language (closely related to the ARM assembly language);
- Automatable verification, through program analysis, of blue/-green separation for bitflips in data registers;
- Development of formally verified, software-only fault tolerance solutions, i.e., "gadgets";
- A tool/methodology for statistical modelling of software fault detection using SMC.

In particular, the formalisation of language, semantics, and fault models, using only elementary and well-known techniques, represents a minor technical contribution but, at the same time, is a significant and non-trivial stepping stone for bringing these techniques to bear on realistic platforms and systems.

This work was partially funded by the SENSATION project (EU FP7 grant number 318490).

[1]http://www.space.aau.dk/aausat3/

[2]http://en.wikipedia.org/wiki/AAUSAT3

[3]http://www.space.com/9585-nasa-revive-cassini-spacecraft-saturn.html

[4]http://aviationweek.com/blog/dragons-radiation-tolerant-design

II. Blue/Green Programs: An Example

In this section, we show an example program that is vulnerable to a bitflip fault and we explain and illustrate the blue/green encoding that has been mentioned several times in the introduction.

Consider the following program snippet from a (hypothetical) control program for a cubesat. The exact syntax and semantics of the program will be explained in Section IV. For safety reasons the control program includes a command to shut down the entire system in case of emergency. The snippet shows where the control program branches to the code implementing the shutdown protocol:

```
LDR r1, 0xFEEDFEED ; get command
MOV r2, 12345   ; move shutdown code to r2
CMP r1, r2      ; compare r1 and r2
BEQ "shutdown"  ; jump to "shutdown" branch
```

First the command is retrieved from the command buffer, e.g., through memory mapped I/O; it is then compared to the (hardwired) command code for shutdown, here ‘12345’. If they match, control is transferred to the actual shutdown code.

Even in this short program, a bitflip in one of the data registers or in one of the flags used for control flow can have unfortunate consequences: if the entered command (stored in register r1) only differs from the shutdown command by one bit, a bitflip in register r1 could be misinterpreted as the shutdown command. Similarly, a bitflip in register r2 immediately before the comparison operation could again lead to the input command to be misinterpreted. Finally, if the status flag indicating the result of the comparison instruction is flipped immediately after the comparison, it would again lead to an unintended shutdown.

One solution to this problem is to use *blue/green encoding*, an approach where all critical values are computed by two independent threads [6], [3], called the blue thread and the green thread respectively, and only if the results computed by both threads agree, critical actions such as storing values in the heap or branching are performed. Converting the above program to use blue/green encoding could result in the following program snippet:

```
LDR r1, 0xFEEDFEED ; get command
MOV r3, 12345      ; move shutdown code to r3
LDR r2, 0xFEEDFEED ; get command
MOV r4, 12345      ; move shutdown code to r4
BEQ_BG r1, r3, r2, r4, "shutdown"
```

Note that the input command is retrieved twice from the heap and two different registers r3 and r4 are loaded with the proper command code ‘12345’ and, finally, the special blue/green instruction ‘BEQ_{BG}’ is used to perform two atomic comparisons in parallel, one for the blue thread and one for the green thread. In this way, a single bitflip in a data register will either be detected and acted upon or ignored if it does not influence the final result.

III. Related Work

The effect of single-event upsets is well known [7], [8], [9], [10], and the fault rate increases by about 8% per generation [2]. The majority of SEUs involve a single bit flipping [7], but varies, e.g., with processor layout and data patterns ([10] finds that a flip from 1 to 0 is three times more likely than from 0 to 1). A number of solutions exist using specialised hardware, e.g., radiation-hardened processors [9] or hardware duplication [5], but these are prohibitively expensive for many applications such as student satellites.

A tempting solution is therefore to use software-only solutions for error detection [11], [6], [12]. Oh et al. [6] propose to use duplicated instructions under a fault model of one a priori permanent bitflip in the program, and to use basic block signatures [11] under a fault model of one a priori permanent corrupted branch instruction in the program. Both methods reduce undetected errors by an order of magnitude, but do not eliminate them. In our fault model the bitflip can happen at any time during execution, and is indeed transient. Nicolescu and Velazco [13] use a C-to-C program transformation to derive a hardened program (blue/green variables, basic block signatures, and duplicated checks before branching) and finds that under radiation testing the hardened program experiences more upsets (due to longer execution time), but reduces the undetected errors by a factor of 3.2. SWIFT [12] is an optimised method, based on [6], that succeeds in eliminating all undetected errors arising from single bitflips in data registers and flags in benchmarks.

While the above cited methods succeed in reducing the error rates, the execution platform and fault models are only specified informally, making it impossible to formally verify the absence of a certain fault under some method, as we have done in Section VI and VII. In [3] Perry et al. formally specify a language, fault model and a type system to guarantee that well-typed programs are indeed fault tolerant under the chosen fault model of corruption of a single register. In [14] the work is extended to reason about control-flow errors. The execution platform however has a number of features not found in contemporary processors: duplicated blue/green PC register to detect errors in the PC [3], and support for catching jumps to the middle of basic blocks [14]. Meola and Walker [15] use a logic based on separation logic to prove imperative programs correct under formal fault models, for a while-like language. In [16] a framework using symbolic execution and model checking is presented, which allows to exhaustively enumerate all faults under a formal fault model on a given program. The fault model considered is only single word corruption, and the programs are re-written to a generic assembly language – thus making it hard to incorporate fault models of bitflips in instruction encoding, as considered by [6] and which we consider in Section VIII.

IV. TinyARMs

In this section we define the syntax and formal semantics for our target language: *TinyARM*. The TinyARM language is a small core language that aims at capturing the essential features of the ARM assembly language [5]. In particular, the instruction set contains no conditionals, at least not in the traditional sense. Instead, individual instructions can be executed conditionally, depending only on the value of one or more status flags that can be modified in a number of ways. For a more comprehensive formalisation of the ARM semantics we refer to [17]. While TinyARM defines only a small number of core instructions, we believe that these are representative for the kinds of instructions found in the full ARM instruction set and a further small step towards bridging the gap between the abstract languages used for formalisation, e.g., the typed assembly language used in the seminal work by Perry et al. [14], [3], and the concrete assembly languages used in [6], [11], [12] for example.

[5]Such as the Hercules chip, with duplicated processors running in lockstep: http://www.ti.com/lsds/ti/microcontroller/safety_mcu/overview.page.

As mentioned in the introduction we will define two variants of TinyARM sharing a common core. The first variant is the standard language with the expected instructions and semantics. The other is a "blue/green" variant of the language which incorporates special instructions for branching and heap manipulation that perform extra safety checks atomically.

Core Language. Values are taken to be 32-bit integers and we explicitly assume that all values are encoded as binary numbers (in a non-specified but consistent encoding): $\mathsf{Val} = \mathbb{B}^{32}$ where $\mathbb{B} = \{0,1\}$ and $\mathbb{B}^{32} = [0..31] \to \mathbb{B}$. The set of values subsumes the set of heap addresses, but for presentation purposes we introduce a separate name for these: $\mathsf{Addr} = \mathsf{Val}$. The ARM architecture defines 16 registers: 13 general purpose registers (denoted r_0 to r_{12}) as well as three *control registers* for holding the stack pointer, the link register (used for function calls), and the program counter respectively. The control registers can be accessed like normal registers but this may lead to undefined results. Since we do not model function calls in TinyARM, we omit the stack pointer and link register and include only the program counter as control register: $\mathsf{DataRegister} = \{r_0, \ldots, r_{12}\}$, $\mathsf{ControlRegister} = \{r_{pc}\}$, and $\mathsf{Register} = \mathsf{DataRegister} \cup \mathsf{ControlRegister}$. Both data- and control-registers hold values: $\mathsf{DataRegisters} = \mathsf{DataRegister} \to \mathsf{Val}$, $\mathsf{ControlRegisters} = \mathsf{ControlRegister} \to \mathsf{Val}$, and $\mathsf{Registers} = \mathsf{Register} \to \mathsf{Val}$.

As already noted, one of the characteristics of the ARM assembly language is the lack of instructions implementing traditional conditionals. Instead most instructions include a *condition field* that specifies for which values of the *condition code flags* a particular instruction should be executed. On the ARM platform there are four condition code flags, called the *Negative*, *Zero*, *Carry*, and *oVerflow* flag respectively which we all model in TinyARM. Each flag can be set to either 0 (false) or 1 (true): $\mathsf{Flag} = \{f_N, f_Z, f_C, f_V\}$ and $\mathsf{Flags} = \mathsf{Flag} \to \mathbb{B}$. With the flags defined, we can now specify the different condition codes for conditional execution: $\mathsf{ConditionCode} = \{\texttt{EQ}, \texttt{NE}, \ldots, \texttt{AL}\}$. The semantics of condition codes, $\chi \in \mathsf{ConditionCode}$, is given by the *cond*-function. It is defined such that $cond(\chi, (f_N, f_Z, f_C, f_V))$ is true iff the relevant flags satisfy the condition χ, e.g., $cond(\texttt{EQ}, (f_N, f_Z, f_C, f_V)) = (f_Z = 1)$. The '`AL`' condition code corresponds to unconditional execution of the instruction. Instructions with a condition code that does not evaluate to true during execution will be treated as a '`NOP`'.

The effect of arithmetic operations on the condition flags is formalised through a family of functions, one for each arithmetic operator '*op*': 'flags_{op}'. E.g., for the addition operator and $v_1, v_2 \in \mathsf{Val}$, $\mathit{flags}_{\texttt{ADD}}(v_1, v_2)(f_Z) = 1$ if $v_1 + v_2 = 0 \pmod{2^{32}}$, and 0 otherwise.

The common core of our two TinyARM variations shares the goals of the *storeless basic block* defined in [6]. Thus, there are no instructions for branching or modification of the heap, these will instead be introduced in Section IV. Note that heap *loads* are still part of the common core since these do not result in observable differences in the heap. This leaves the following instruction set for the common core:

$\mathsf{Instr}_{\mathsf{Core}}$	$::=$	$\texttt{MOV}\chi\, x\texttt{,}\ v$	store value v in x
	$\mid$	$\texttt{MOV}\chi\, x\texttt{,}\ y$	store content of y in x
	$\mid$	$\texttt{OP}\chi\, x\texttt{,}\ y\texttt{,}\ z$	do "`OP`" on y and z, store in x
	$\mid$	$\texttt{OPS}\chi\, x\texttt{,}\ y\texttt{,}\ z$	like "`OP`" but also set flags
	$\mid$	$\texttt{CMP}\chi\, x\texttt{,}\ y$	compare x and y, set flags
	$\mid$	$\texttt{LDR}\chi\, x\texttt{,}\ a$	load x from heap address a
	$\mid$	$\texttt{LDR}\chi\, x\texttt{,}\ y$	load x from heap address in y

$$\frac{\begin{array}{c} P(R(r_{pc})) = \texttt{OPS}\chi\, x\texttt{,}\ y\texttt{,}\ z \qquad cond(\chi, F) \\ F' = \mathit{flags}_{op}(R(x), R(y)) \end{array}}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}[x \mapsto R(y)\ op\ R(z)], F'\rangle}$$

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

$$\frac{\begin{array}{c} P(R(r_{pc})) = \texttt{CMP}\chi\, x\texttt{,}\ y \qquad cond(\chi, F) \\ F' = \mathit{flags}_{\texttt{CMP}}(R(x), R(y)) \end{array}}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, F'\rangle}$$

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

Figure 1. Excerpt of operational semantics for $\mathsf{Program}_{\mathsf{Core}}$

$$\frac{P(R(r_{pc})) = \texttt{B}\chi\, x \qquad cond(\chi, F)}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R[r_{pc} \mapsto R(x)], F\rangle}$$

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

Figure 2. Excerpt of semantics for observable actions in $\mathsf{Program}_{\mathsf{Obs}}$

where $\chi \in \mathsf{ConditionCode}$, $x, y, z \in \mathsf{DataRegister}$, $a \in \mathsf{Addr}$, and $v \in \mathsf{Val}$. Note that all of the above instructions are parameterised on a condition code allowing them to be executed conditionally.

A program is then formalised simply as a map from addresses to instructions: $\mathsf{Program}_{\mathsf{Core}} = \mathsf{Addr} \to \mathsf{Instr}_{\mathsf{Core}}$ and the heap memory as a map from addresses to values: $\mathsf{Heap} = \mathsf{Addr} \to \mathsf{Val}$. We then formalise the configurations of our structural operational semantics for the core language as follows $\mathsf{Conf}_{\mathsf{Core}} = \mathsf{Program}_{\mathsf{Core}} \times \mathsf{Heap} \times \mathsf{Registers} \times \mathsf{Flags}$. The semantic configurations are mostly standard for an operational semantics with the minor exception that we include the program itself in the configurations: this allows us to formalise the effects of bitflips in the program code itself. To the best of our knowledge, this is the first formal semantics incorporating such a fault model. Finally, the semantics of the core language is completed by defining a reduction relation between semantic configurations $C, C' \in \mathsf{Conf}_{\mathsf{Core}}$: $C \Longrightarrow C'$. We let '$\Longrightarrow^n$' and '$\Longrightarrow^*$' denote a reduction sequence of length n and the reflexive and transitive closure of '$\Longrightarrow$' respectively and as a further notational convenience we introduce the following notation for performing updates of a register value: for $z \in \mathbb{Z}$ define $R_{x+z} = R[x \mapsto R(x) + z]$. In particular: $R_{r_{pc}+1}$ denotes updating the program counter to prepare for the next instruction. The reduction rules for the core language are fairly straightforward. An excerpt of the semantics is shown in Figure 1.

Observable Actions: Heap Store and Branching. Here we extend the core language, as defined above, with instructions for branching and for storing values in the heap. As discussed in Section II, these instructions may lead to observable differences in the heaps produced during program execution and/or differences in the termination behaviour. Below, only the instructions that have been added to the core language are shown:

$\mathsf{Instr}_{\mathsf{Obs}}$	$::=$	$\cdots$	(instructions from $\mathsf{Instr}_{\mathsf{Core}}$)
	$\mid$	$\texttt{B}\chi\, a$	branch to address a (goto)
	$\mid$	$\texttt{B}\chi\, x$	branch to address stored in x
	$\mid$	$\texttt{STR}\chi\, x\texttt{,}\ a$	store content of x at address a
	$\mid$	$\texttt{STR}\chi\, x\texttt{,}\ y$	store content of x at address in y

where $\chi \in \mathsf{ConditionCode}$, $x, y \in \mathsf{DataRegister}$, and $a \in \mathsf{Addr}$.

The notion of a program is trivially extended to cover the new instructions: $\mathsf{Program}_{\mathtt{Obs}} = \mathsf{Addr} \rightarrow \mathsf{Instr}_{\mathtt{Obs}}$. Similarly, the semantic configurations only change minimally: $\mathsf{Conf}_{\mathtt{Obs}} = \mathsf{Program}_{\mathtt{Obs}} \times \mathsf{Heap} \times \mathsf{Registers} \times \mathsf{Flags}$. An excerpt of the semantic rules for the observable actions are shown in Figure 2.

Blue/Green Encoded Heap Store and Branching. Finally, we define the *blue/green* variant of our TinyARM language. The added instructions that make up this variant provide explicit support for the blue/green encoding described in [3] and are based on ideas in [6].

In the following, the x and y registers are "green" registers, while the x' and y' registers are "blue" registers. The content of blue and green registers should be computed separately and yield the same result. This comparison and execution of the instruction is assumed to be performed *atomically* and *free of bitflips*. In Section VII we show how (the effect of) some of these instructions can be implemented without hardware support. The instructions making up the blue/green TinyARM variant are as follows:

$$
\begin{array}{lcll}
\mathsf{Instr}_{\mathtt{BG}} & ::= & \cdots & \text{(instr. from } \mathsf{Instr}_{\mathtt{Core}}\text{)} \\
 & \mid & \mathtt{B}_{\mathtt{BG}}\ x\mathtt{,}\ x' & \text{branch to address in } x \\
 & \mid & \mathtt{BEQ}_{\mathtt{BG}}\ x\mathtt{,}\ y\mathtt{,}\ x'\mathtt{,}\ y'\mathtt{,}\ a & \text{atomic cmp. and branch} \\
 & \mid & \mathtt{STR}_{\mathtt{BG}}\ x\mathtt{,}\ x'\mathtt{,}\ a & \text{store content of } x \text{ at } a \\
 & \mid & \mathtt{STR}_{\mathtt{BG}}\ x\mathtt{,}\ y\mathtt{,}\ x'\mathtt{,}\ y' & \text{store content of } x \text{ at } y
\end{array}
$$

where $x, x', y, y' \in \mathsf{DataRegister}$ and $a \in \mathsf{Addr}$. Again the notion of a program is trivially extended to cover the new instructions: $\mathsf{Program}_{\mathtt{BG}} = \mathsf{Addr} \rightarrow \mathsf{Instr}_{\mathtt{BG}}$. If the comparison of a blue and a green register fails in one of the added instructions, the entire computation fails. This is reflected in the semantic configurations by adding an explicit fail-state: $\mathsf{Conf}_{\mathtt{BG}} = (\mathsf{Program}_{\mathtt{BG}} \times \mathsf{Heap} \times \mathsf{Registers} \times \mathsf{Flags}) + \{\mathtt{FAIL}\}$. Since we focus on fault-detection, and not fault-recovery, we simply model failure as a terminal state in the semantics. Figure 3 shows the semantic rules for the added blue/green instructions. For brevity, we tacitly assume that the last rule, the "fail rule", is only invoked if none of the preceding rules for blue/green instructions can be used for a specific instruction.

In the case of the $\mathtt{BEQ}_{\mathtt{BG}}$ instruction two copies of two values are compared. They must all four be equal for the equality condition to be satisfied. Inequality however is often preserved by bitflips. We exploit *logical masking* to safely ignore an error that does not change the outcome of the branch [12]. Only if one pair of x/y values (say, the blue pair) is mutually equal and the other is not, the computation fails.

V. Fault Models and Fault Tolerance

The occurrence of faults during program execution is modelled explicitly as special semantic rules defining the so-called *fault model*. This simplifies both formalisation and later analysis. In order to track the faults that occur during execution, the semantic "error rules" are annotated with the kind of fault (see below for details) that has occurred: $C \Longrightarrow_{\phi} C'$ where $\phi \in \mathcal{F}$ denotes the fault that has occurred and $\mathcal{F}$ is the set of faults that are possible in the given fault model, formally defined below.

Figure 4 shows the semantic formalisation of several fault types well-known from the literature. The first two error rules, DSEU and CSEU, are concerned with a single-event upset (SEU) in a processor register. This type of fault is divided over two rules since the consequences of bitflips in data and control registers differ widely.

$$\frac{P(R(r_{pc})) = \mathtt{B}_{\mathtt{BG}}\ x\mathtt{,}\ x' \qquad R(x) = R(x')}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R[r_{pc} \mapsto R(x)], F\rangle}$$

$$\frac{\begin{array}{c}P(R(r_{pc})) = \mathtt{BEQ}_{\mathtt{BG}}\ x\mathtt{,}\ y\mathtt{,}\ x'\mathtt{,}\ y'\mathtt{,}\ a \\ R(x) = R(y) = R(x') = R(y')\end{array}}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R[r_{pc} \mapsto a], F\rangle}$$

$$\frac{\begin{array}{c}P(R(r_{pc})) = \mathtt{BEQ}_{\mathtt{BG}}\ x\mathtt{,}\ y\mathtt{,}\ x'\mathtt{,}\ y'\mathtt{,}\ a \\ R(x) \neq R(y) \qquad R(x') \neq R(y')\end{array}}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H, R_{r_{pc}+1}, F\rangle}$$

$$\frac{\begin{array}{c}P(R(r_{pc})) = \mathtt{STR}_{\mathtt{BG}}\ x\mathtt{,}\ y\mathtt{,}\ x'\mathtt{,}\ y' \\ R(x) = R(x') \qquad R(y) = R(y')\end{array}}{\langle P, H, R, F\rangle \Longrightarrow \langle P, H[R(y) \mapsto R(x)], R_{r_{pc}+1}, F\rangle}$$

$$\frac{P(R(r_{pc})) = instr \qquad instr \in \mathsf{Instr}_{\mathtt{BG}}}{\langle P, H, R, F\rangle \Longrightarrow \mathtt{FAIL}}$$

Figure 3. Semantics for the blue/green instructions in $\mathsf{Program}_{\mathtt{BG}}$

$$\frac{v = R(x) \qquad v' \equiv_1 v \qquad x \in \mathsf{DataRegister}}{\langle P, H, R, F\rangle \Longrightarrow_{\text{DSEU}} \langle P, H, R[x \mapsto v'], F\rangle}\ \text{(DSEU)}$$

$$\frac{v = R(x) \qquad v' \equiv_1 v \qquad x \in \mathsf{ControlRegister}}{\langle P, H, R, F\rangle \Longrightarrow_{\text{CSEU}} \langle P, H, R[x \mapsto v'], F\rangle}\ \text{(CSEU)}$$

$$\frac{F' = F[f \mapsto \overline{F(f)}]}{\langle P, H, R, F\rangle \Longrightarrow_{\text{FSEU}} \langle P, H, R, F'\rangle}\ \text{(FSEU)}$$

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

Figure 4. Fault semantics

We formalise when two values, or rather their respective binary encodings, differ by only one bit; essentially corresponding to the *Hamming distance* used in [6]. For $b_1, b_2 \in \mathbb{B}^{32}$ define $b_1 \equiv_1 b_2$ iff $\exists i\colon \forall j\colon b_1(j) \neq b_2(j) \iff j = i$. This can obviously be extended to differences in any number of bits. The FSEU rule deals with a SEU in a condition flag. Let $F \in \mathsf{Flags}$ and $f \in \mathsf{Flag}$ and define $\overline{F(f)} = 1 - F(f)$. We further assume the existence of a function that maps instructions to their binary encoding $encode\colon \mathsf{Instr} \rightarrow \mathbb{B}^{32}$. The single-instruction corruption (SIC) fault considers a SEU in the encoding of an instruction. Since we generally assume that memory is protected, the bitflip happens in the currently executing instruction and the program text remains unchanged. This and the CSEU type of fault are treated using statistical model-checking in Section VIII.

Similar to the "ordinary" semantics, we let '$\Longrightarrow^{n}_{\phi}$' and '$\Longrightarrow^{*}_{\phi}$' denote reduction sequences of length n and the reflexive and transitive closure of '$\Longrightarrow$' with a single fault step of the form $C \Longrightarrow_{\phi} C'$ somewhere in the sequence. In this case, we call ϕ the *fault trace* of the semantic reduction sequence.

A fault model can now be formalised simply as the set of fault traces that can occur:

Definition 1 (Fault Model)**.** *A* fault model, $\mathcal{F}$ *consists of the set of possible fault traces:* $\mathcal{F} = \{\phi_i\}_i$.

Various authors study "the SEU fault model", among others [3],

[6], [12]. While their definitions vary we define it to mean one bitflip in either a general purpose register or in one of the condition flags, but *not* in the register holding the program counter. For ease of reference we define the formal SEU and SIC fault models here: $\mathcal{F}_{\text{SEU}} = \{\text{DSEU}, \text{FSEU}\}$ and $\mathcal{F}_{\text{SIC}} = \{\text{SIC}\}$.

A. Fault Tolerance

Having defined what it means for a fault to occur in a program run, we now turn to formalising what it means for a program to be *fault tolerant*. Here, we follow the same approach as [3] and define a program to be fault tolerant, with respect to a given fault trace ϕ, if it is able to either detect that an error has occurred and fail in a controlled manner or else continue working and yield the same or equivalent results. In anticipation of later sections, we parameterise our notion of fault tolerance on the equivalence ($\equiv$) used to determine if the results are similar "enough" and extend the definition to fault models:

Definition 2 (ϕ/$\equiv$-tolerant)**.** *Let ϕ be a fault trace and $P \in \mathsf{Program}_{\mathtt{BG}}$ with $C = \langle P, H, R, F\rangle \in \mathsf{Conf}_{\mathtt{BG}}$ such that $C \Longrightarrow^n C'$ for some n, then P is* ϕ/$\equiv$-tolerant *if and only if one of the following holds: (1) if $C \Longrightarrow_\phi^{n+1} C''$ then $C' \equiv C''$; or (2) $\exists m\colon m \leq n$ such that $C \Longrightarrow_\phi^m$* $\mathtt{FAIL}$.

Definition 3 ($\mathcal{F}$/$\equiv$-tolerant)**.** *Let $\mathcal{F} = \{\phi_i\}_i$ be a fault model and $P \in \mathsf{Program}_{\mathtt{BG}}$, then P is* $\mathcal{F}$/$\equiv$-tolerant *if and only if it is ϕ_i/$\equiv$-tolerant for every fault trace in $\{\phi_i\}_i$.*

One obvious choice of equivalence is that of equality, i.e., $\mathcal{F}$/$=$-tolerance. However, as we shall see in the next section, requiring equality is unnecessarily strict. Intuitively it should be enough to require that only the registers that are actually needed to produce the end result are identical.

VI. Verified Blue/Green Separation

In this section we show how the formalisation of language semantics and fault models in the previous sections can be used to develop, and formally prove correct, a static analysis that guarantees proper blue/green separation in a blue/green-program (defined in Section IV). We further show that this is sufficient to make the program $\mathcal{F}_{\text{SEU}}$-tolerant (see Section V-A).

The basic idea in our analysis is to assign a colour to every data-register, either blue or green, and then to track the "colour dependency" of all registers and flags at every program point, i.e., track what colour of registers the content of a given register depends on. This information can then be used to verify that a register of a given colour only depends on other registers of the same colour and therefore be completely independent of registers of the other colour irrespective of any bitflips in those registers.

A. Flow Logic Specification

For the analysis, we use the specification-oriented *Flow Logic* approach of Nielson and Nielson [18] which sets forth judgements for an acceptable analysis result. The essential idea in our analysis is to track the colour of the registers that has influenced the value of a given register (or flag) for every program point. Thus, abstract values are taken from the following complete lattice: $\widehat{\mathsf{Val}} = (\mathcal{P}(\{\mathsf{B}, \mathsf{G}\}), \subseteq)$.

As an excerpt, we present the Flow Logic judgement for the '`OPS`'-instruction, where $\hat{R}\colon \mathsf{PC} \to \mathsf{DataRegister} \to \widehat{\mathsf{Val}}$ and $\hat{F}\colon \mathsf{PC} \to \widehat{\mathsf{Val}}$ are the analysis results for the register and flag valuations, respectively:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\hat{R}, \hat{F}) \models pc : \mathtt{OPS}\chi\ x,\ y,\ z} \\
\text{iff} & \hat{R}(pc)(y) \cup \hat{R}(pc)(z) \subseteq \hat{R}(pc+1)(x) \\
& \hat{R}(pc)(y) \cup \hat{R}(pc)(z) \subseteq \hat{F}(pc+1) \\
& \chi \neq \mathtt{AL} \implies \hat{F}(pc) \subseteq \hat{R}(pc+1)(x) \\
& \chi \neq \mathtt{AL} \implies \hat{F}(pc) \subseteq \hat{F}(pc+1) \\
& \hat{R}(pc) \subseteq_{\{x\}} \hat{R}(pc+1)
\end{array}$$

The judgement models the semantics closely: first, (in the first line), the colour of the result register, x, depends (at the next program point $pc+1$) on the colours of the operand registers y and z (at the current program point pc). Similarly (in the second line), the condition flags at the next program point depend on the colours of the operand registers at the current program point (the five lines in the condition are joined by an implicit conjunction). The third line handles conditional execution: if the condition code indicates conditional execution (anything but the '`AL`' condition code), then the colours that have influenced the flags at the current program point also, potentially, influence the content of result register at the next program point. The fourth line does the same, only for the condition flags. Finally (in the fifth line), all the abstract values of all the registers that are not modified by this instruction are copied to the next program point. The judgement describes subset relations rather than equality because, in the case of jumps, several judgements can impose requirements on the values at a single program point.

Based on the blue/green information flow analysis, we now define the notion of static blue/green separation that can be used to statically guarantee that a program implements proper separation between the blue and the green threads of computation, which will be used below to show that a properly separated blue/green program is also fault tolerant with respect to the SEU-fault model:

Definition 4 (Static B/G Separation)**.** *Let $P \in \mathsf{Program}_{\mathtt{BG}}$ such that $(\hat{R}, \hat{F}) \models P$, then P is said to be* statically B/G separated (with respect to $(\hat{R}, \hat{F})$) *if and only if for all $pc \in \mathrm{dom}(P)$ it holds that $\hat{F}(pc) \subset \{\mathsf{B}, \mathsf{G}\}$ and $\forall r \in \mathsf{DataRegister}\colon \hat{R}(pc)(r) \subset \{\mathsf{B}, \mathsf{G}\}$.*

While the specification of our blue/green information flow analysis seems rather abstract, it is straightforward to convert the formulae of the specification, e.g., into Datalog [19] and use a corresponding solver to obtain an implementation of our analysis.

B. Formal Properties

We now formally establish the correctness of our analysis. This is done through a series of technical lemmas of which we only include a few of the most important for illustration purposes. We define $R_1 \equiv_B R_2$ to mean equality on the set of blue registers and $R_1 \equiv_G R_2$ to mean equality on the set of green registers.

Lemma 1. *Let $P \in \mathsf{Program}_{\mathsf{Core}}$ with $m = |P|$, $(\hat{R}, \hat{F}) \models P$ and let P be statically B/G separated (wrt. $(\hat{R}, \hat{F})$), then if $\langle P, H, R_1, F\rangle \Longrightarrow^m \langle P, H, R_1', F_1'\rangle$ and $\langle P, H, R_2, F\rangle \Longrightarrow^m \langle P, H, R_2', F_2'\rangle$ it holds that $R_1 \equiv_B R_2 \implies R_1' \equiv_B R_2' \wedge (F_1' = F_2' \vee \hat{F}(m) \supseteq \{G\})$ and symmetrically for $R_1 \equiv_G R_2$.*

Lemma 2. *Let $P \in \mathsf{Program}_{\mathtt{BG}}$, $\phi \in \mathcal{F}_{SEU}$ such that $\langle P, H, R, F\rangle \Longrightarrow \langle P', H', R', F'\rangle$, $\langle P, H, R, F\rangle \Longrightarrow_\phi C''$, $(\hat{R}, \hat{F}) \models P$, and P is statically B/G separated (wrt. $(\hat{R}, \hat{F})$). Then either $C'' =$* $\mathtt{FAIL}$ *or $C'' = \langle P', H'', R'', F''\rangle$ with $H' = H''$ and $R' \equiv_{B/G} R''$ and $F' = F''$.*

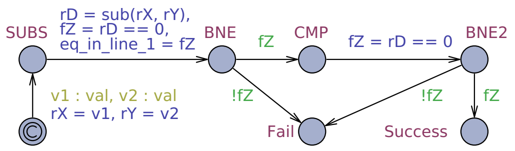

Figure 5. Model of the robust assert gadget

By splitting a blue/green program into *basic blocks*, consisting only of core instructions and ending with a blue/green instruction (similar to the storeless basic blocks of [6]), and using the above lemmas on these basic blocks, the following Theorem can be proved by induction in the length of the reduction, establishing the SEU-tolerance of a blue/green program that is statically B/G separated:

Theorem 1. *Let* $P \in \mathsf{Program}_{\mathtt{BG}}$ *and* $(\hat{R}, \hat{F}) \models P$ *such that* P *is statically B/G separated (wrt.* $(\hat{R}, \hat{F})$*), then* P *is SEU-tolerant.*

VII. Gadgets

Since the blue/green instructions require special hardware support we introduce *gadgets* instead. Gadgets are small blocks of core language instructions that fulfill certain functions and detect if an error occurs during their own execution. We start by considering a gadget that is used as a building block for the gadgets that realise the functions of the special blue/green instructions.

A. Robust Assert

The first gadget robustly asserts that two data registers contain the same value. By "robust" we mean that if an error occurs while the gadget is executing, the error is guaranteed to be detected. The gadget consists of four core language instructions, and falling through the gadget signifies that the assertion held:

```
1  SUBS rD, rX, rY ; rD = rX - rY, fZ
   = (rD == 0)
2  BNE "fail"     ; fail if fZ == 0
3  CMP rD, #0     ; fZ = (rD == 0)
4  BNE "fail"     ; fail if fZ == 0
5  ...            ; success
```

Lemma 3. *Under* $\mathcal{F}_{\mathit{SEU}}$*, line 5 (success) is reached only if* `rX` *and* `rY` *contain the same value when line 1 is executed.*

Lemma 4. *Under* $\mathcal{F}_{\emptyset}$*, line 5 is reached if and only if* `rX` *and* `rY` *contain the same value when line 1 is executed.*

These lemmas are verified by model checking as explained in the next section.

B. Model Checking Gadgets

We model the gadgets in the UPPAAL model checker [20] as follows: Registers are global variables, and two automata are used for the gadget and the bitflip, respectively. For the robust assert gadget, these two automata are shown in Figures 5 and 6, described below. The registers are modelled after these considerations: Two identical values contained in two registers will always become unequal if one of their bits is flipped. For two different values, two situations are possible: With a Hamming distance of one, a bitflip may or may not render the two values equal. With a Hamming distance greater than one, no flip will make them equal. This implies that we only

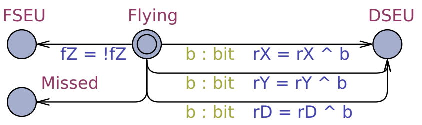

Figure 6. Proton model

need enough detail to model Hamming distances of 0, 1, and >1. Two bits are sufficient for this so we model the registers as 2-bit values. The *Zero* condition flag is currently the only relevant flag and it is represented by a boolean variable. The initial location, marked by circle, of the gadget automaton in Figure 5 is also committed (marked by a 'C') meaning that it must be left immediately, specifically before a bitflip occurs. The initial location leads to the first instruction of the gadget. The *select* statement on the edge in effect describes a transition for each possible assignment of values to the registers such that the model checking considers all scenarios wrt. Hamming distances.

The model has a location for each program point and an edge for each possible control flow. On the edges the register values are updated as each instruction prescribes. The edge executing the first instruction of the gadget also records the state of equality between the two registers. The two outgoing edges from each conditional branch are guarded by the branch condition and its negation.

The initial location of the proton automaton in Figure 6 has an outgoing edge for each bit that can be flipped. Each edge leads to a terminal location such that the proton may cause at most one bitflip. The interleaving implied by the parallel composition of the two automata and the exhaustive exploration done by the model checker lead to every possible bitflip being tested, including the proton striking before the execution of the first line of the gadget. The query `A[] (Gadget.Success imply eq_in_line_1)`, verifies Lemma 3. Lemma 4 is verified by the query `A[] ((eq_in_line_1 and Proton.Missed and deadlock) imply Gadget.Success)` and Lemma 3.

Model checking these queries takes less than a second, and the gadget can now be used any number of times in a program. Scalability is thus not an issue.

C. Blue/Green Branch

We now introduce the blue/green branch gadget, the goal of which is to replace the $\mathtt{BEQ}_{\mathtt{BG}}$ x, y, x', y', a instruction. It is a three-way branching gadget that jumps to a hardcoded destination if the argument registers contain the same value at the beginning of the execution of the gadget. If the values differ consistently (the blue/green copies are identical), control falls through as in a standard conditional branch instruction. On any inconsistency it jumps to the "fail" location.

```
  SUBS rD, rX, rY
  BNE "NEQ_bias"
EQ_bias:
  CMP rD, #0
  BNE "fail"
  SUBS rD, rX', rY'
  BNE "fail"
  CMP rD, #0
  BNE "fail"
  B   "equals"
NEQ_bias:
  CMP rD, #0
  BEQ "fail"
  SUBS rD, rX', rY'
  BEQ "fail"
  CMP rD, #0
  BEQ "fail"
  ... ; success
```

The gadget chooses a "bias" based on the comparison between `rX` and `rY` and confirms that registers `rX'` and `rY'` compare in the same way. Each of the three comparisons are also checked for consistency wrt. the duplicated information written to register `rD` using the technique from the robust assert gadget.

Theorem 2. *Under $\mathcal{F}_{SEU}$:*

- *Line 10 is reached only if* $\texttt{rX} = \texttt{rY} = \texttt{rX}' = \texttt{rY}'$ *in line 1,*
- *Line 18 is reached only if* $\texttt{rX} \neq \texttt{rY} \wedge \texttt{rX}' \neq \texttt{rY}'$ *in line 1.*

Theorem 3. *Under $\mathcal{F}_{\emptyset}$:*

- *Line 10 is reached if and only if* $\texttt{rX} = \texttt{rY} = \texttt{rX}' = \texttt{rY}'$ *in line 1,*
- *Line 18 is reached if and only if* $\texttt{rX} \neq \texttt{rY} \wedge \texttt{rX}' \neq \texttt{rY}'$ *in line 1.*

Note that the condition $\texttt{rX} \neq \texttt{rY} \wedge \texttt{rX}' \neq \texttt{rY}'$ does not require the blue/green values to be consistent ($\texttt{rX} = \texttt{rX}' \wedge \texttt{rY} = \texttt{rY}'$). As discussed in Section IV, logical masking allows us to safely ignore an error that does not affect the behaviour of the program.

VIII. Quantifying Risk with SMC

For some fault models it is very challenging to give guarantees using only software, simply because they cause the software to lose control of the execution: consider the fault models CSEU (a bitflip in a control register, such as the program counter), and SIC (a bitflip in the encoding of the instruction being executed) from Figure 4. Under such fault models we would still like to quantify the risk of a bitflip causing faulty execution, in order to reason about the relative effectiveness of the proposed fault tolerance solutions, such as blue/green programs.

We propose to use the simulation-based statistical model checking (SMC) [4] to model the program and fault model, and simulate the impact faults can have on the execution. The use of SMC enables formal modelling of faults and exact impact; because the entire system state can be altered, faults can be injected precisely compared to physical experiments where the fault model is not under precise control. In addition SMC is faster than a cycle-accurate simulator, as the model time can be accelerated, or slowed down, arbitrarily as needed.

We have developed a toolchain for extracting an UPPAAL automata model from an ARM binary. The UPPAAL model is subsequently analysed using the UPPAAL SMC tool [4].

The extracted models have the following components: *(1)* A copy of the program and registers that are vulnerable to bitflips, called P_1; *(2)* a copy of the program and registers not affected by bitflips, called P_2; this copy is used to check whether the observable behaviours of P_1 and P_2 are equivalent; *(3)* a component modelling and observing the main memory, with which P_1 and P_2 interact, called *MemoryObserver*. This component monitors whether the observable behaviour of P_1 is different from that of P_2, in terms of termination or stores. We exploit the fact that memory content will be the same for both copies of the program, as long the same observable behaviour is seen, to only have one copy of main memory; *(4)* a *Proton* component interacting with P_1 according to the fault model. The program models P_1 and P_2 are generated automatically from ARM binaries, and include the semantics of the instructions of the program.

The observer is always in one of four states: *(1) Running* meaning that up to this point no observably different behaviour has occurred. *(2) Terminated* meaning that both programs have terminated. *(3) Unsafe* meaning that P_1 behaved observably different from P_2. *(4) Fault detected* meaning that P_1 detected a fault, and aborted execution; explicitly by detection using blue/green encoding or implicitly by causing a processor exception (executing an illegal instruction, accessing unmapped memory, etc.).

In general the probability of a bitflip should be uniformly distributed over the entire runtime of the program; the probability of a proton hitting should be the same for any moment in time. This poses a slight problem in the modelling because the runtime of the program is a priori not known. Therefore the following phases are used for any trace: *(1)* The *input* for the program is chosen non-deterministically, from a uniform distribution. *(2)* P_2 is run using this input, and recording the execution time, *exectime*. *(3)* The *Proton* component chooses a time instance to strike, uniformly from the range $[0, \mathit{exectime}]$. *(4)* P_1 and P_2 are run in parallel on the same input, while being observed by *MemoryObserver*. *(5)* At some point during execution *Proton* strikes, possibly altering P_1's execution. This implements the fault model that exactly one bitflip occurs, during any trace.

The statistical model checker runs a specified number of these traces for a specified maximal number of steps, while recording the status of each trace, thus giving a probability of the observer being in a certain state at the end of the run. Table I gives the probabilities calculated for a real-ARM version of the example program from Section II, with 50000 traces simulated for each scenario, for a maximum of 3000 cycles. The still-running traces are caused by a bitflip altering the control-flow of the program and in reality could either terminate or be unsafe at some point in the future execution. Incrementing the number of steps simulated can possibly reduce the number of still-running traces, but in our experience the ratio does not change by doing this.

The exact numbers vary with a number of factors: If many registers are not used by the concrete program (or only used sparingly) many bitflips will be logically masked because the values in the registers are never read – as can be observed partially by the basic program terminating successfully more often as it uses less registers. For a real processor the probability of a bitflip is not the same for all bits [8], [10], which our fault model does not take into account[6]. Therefore the exact numerical values are of little interest – however comparing the numbers for different programs with identical observable behaviour in the absence of bitflips can compare different programs' susceptibility to bitflips causing malfunctions. As an example, the risk for a program can be compared to the risk of the same program under blue/green encoding: as Table I shows, blue/green encoding eliminates all unsafe traces under the DSEU and FSEU fault models – as expected from Theorems 2 and 3.

A. Risk under Aggressive Fault Models: CSEU and SIC

SMC allows us to model the very aggressive fault models of bitflips in a control register, or even bitflips in the encoding of an instruction. Bitflips in the PC register (CSEU) are modelled by control jumping to an unexpected location of the program P_1, or by jumping to a special crash location if the bitflip would result in jumping outside the program memory.

[6]But if real probabilities are known these could easily be incorporated into the model

For modelling the behaviour of a bitflip in the encoding of an instruction (SIC) the possible mutants for each instruction are generated, and included in the model for P_1. We have used the ARM instruction set encoding, as parsed by the GNU Objdump utility. As an example, a `mov r3, 28` instruction can be bitflipped into a `orr r3, r0, 28` instruction, significantly altering the semantics of the instruction. A bitflip typically alters the instruction opcode, alters the destination or source registers, or alters constants in the instruction.

Our results show that blue/green encoding does not eliminate all unsafe behaviour under the CSEU and SIC fault models. However, Table I clearly shows that blue/green encoding reduces the risk of unsafe behaviour under these very aggressive fault models by an order of magnitude.

Table I. RISK ANALYSIS USING SMC OF THE PROGRAM FROM SECTION II. BG INDICATES THE PROGRAM WAS MODIFIED USING BLUE/GREEN ENCODING AND THE GADGETS FROM SECTION VII. THE FAULT MODELS REFER TO FIGURE 4.

Fault	BG	Running	Terminated	Unsafe	Fault
DSEU, FSEU	No	0%	96.35%	3.64%	0%
	Yes	0%	90.39%	0%	9.61%
+CSEU	No	0.10%	90.87%	3.55%	5.47%
	Yes	0.18%	85.30%	0.046%	14.48%
+SIC	No	0.34%	87.86%	5.09%	6.72%
	Yes	0.20%	82.84%	0.11%	16.84%

The modelling of the processor hardware is at present very simple: the hardware is modelled as a one-stage pipeline, with no caches and a memory access time of 1 cycle. However, note that many bitflips that occur in, e.g., a pipelined processor are equivalent to a bitflip before or after execution of an instruction, and thus the simple model will be accurate enough. We especially note that: If each bit of the instruction encoding is read in exactly one pipeline stage, any bitflip during pipelined execution is equivalent to a bitflip before or after execution of the instruction.

IX. CONCLUSION

We have formalised TinyARM, an assembly language close to the ARM language, and several fault models based on bitflips of varying degree of potency: data, flags, control registers or instruction encoding. We have formalised a blue/green variant of TinyARM which includes instructions that facilitate the blue/green encoding but that are not part of the ARM language, and specified a program analysis for verifying proper blue/green separation. Furthermore we have developed a number of *gadgets* that allow the transformation of a blue/green TinyARM program to a program using only core (Tiny)ARM instructions, while still preserving fault tolerance against bitflips in data registers and flags.

Finally, we have used *statistical model checking* to model and quantify the effects of fault models for which absolute guarantees are hard to provide (bitflips in control registers and instruction encoding). Our experiments indicate that the usage of blue/green gadgets can reduce the risk of unsafe execution by several orders of magnitude under the most aggressive fault models.

Future work is to extend our approach in several directions, including fault recovery, gadgets for majority voting, and more detailed modelling of the underlying hardware.

REFERENCES

[1] E. Normand, “Single event upset at ground level,” *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[2] S. Borkar, “Designing reliable systems from unreliable components: The challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov. 2005.

[3] F. Perry, L. W. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker, “Fault-tolerant typed assembly language,” in *Proc. of Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2007, pp. 42–53.

[4] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang, “Time for statistical model checking of real-time systems,” in *Proc. of Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 6806. Springer Verlag, 2011, pp. 349–355.

[5] *ARM Architecture Reference Manual*, ARM Ltd., Jul. 2005, issue I.

[6] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[7] C. I. Underwood, R. Ecoffet, S. Duzeffier, and D. Faguere, “Observations of single-event upset and multiple-bit upset in non-hardened high-density SRAMs in the TOPEX/Poseidon orbit,” in *Radiation Effects Data IEEE Workshop*, 1993, pp. 85–92.

[8] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, “Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline,” in *Proc. of Dependable Systems and Networks (DSN)*, 2004, pp. 61–71.

[9] F. Wang and V. D. Agrawal, “Single Event Upset: An Embedded Tutorial,” in *Proc. of VLSI Design (VLSID)*, 2008, p. 429.

[10] G. M. Swift, F. F. Fannanesh, S. M. Guertin, F. Irom, and D. G. Millward, “Single-event upset in the PowerPC750 microprocessor,” *Nuclear Science, IEEE Transactions on*, vol. 48, no. 6, pp. 1822–1827, 2001.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software implemented fault tolerance,” in *Proc. of Symposium on Code Generation and Optimization (CGO)*, Mar. 2005, pp. 243–254.

[13] B. Nicolescu and R. Velazco, “Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results,” in *Proc. of Design, Automation & Test in Europe (DATE)*, 2003, pp. 20 057–20 063.

[14] F. Perry and D. Walker, “Reasoning about control flow in the presence of transient faults,” in *Proc. of Static Analysis Symposium (SAS)*, ser. Lecture Notes in Computer Science, vol. 5079. Springer Verlag, 2008, pp. 332–346.

[15] M. L. Meola and D. Walker, “Faulty logic: reasoning about fault tolerant programs,” in *Proc. of Programming Languages and Systems (ESOP)*. Springer Verlag, 2010, pp. 468–487.

[16] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, “SymPLFIED: Symbolic program-level fault injection and error detection framework,” in *Proc. of Dependable Systems and Networks (DSN)*, 2008, pp. 472–481.

[17] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, “The semantics of POWER and ARM multiprocessor machine code,” in *Proc. of Workshop on Declarative Aspects of Multicore Programming (DAMP)*. ACM, Jan. 2009, pp. 13–24.

[18] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer Verlag, 1999.

[19] A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli, “Static analysis in datalog extensions,” *J. ACM*, vol. 48, no. 5, pp. 971–1012, Sep. 2001.

[20] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a nutshell,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.