Fault-Tolerant Non-interference

Filippo Del Tedesco, Alejandro Russo, and David Sands

Chalmers University of Technology, Sweden

Abstract. This paper is about ensuring security in unreliable systems. We study systems which are subject to transient faults – soft errors that cause stored values to be corrupted. The classic problem of fault tolerance is to modify a system so that it works despite a limited number of faults. We introduce a novel variant of this problem. Instead of demanding that the system works despite faults, we simply require that it remains secure: wrong answers may be given but secrets will not be revealed. We develop a software-based technique to achieve this fault-tolerant non-interference property. The method is defined on a simple assembly language, and guarantees security for any assembly program provided as input. The security property is defined on top of a formal model that encompasses both the fault-prone machine and the faulty environment. A precise characterization of the class of programs for which the method guarantees transparency is provided.

1 Introduction and Overview

Transient faults occur in hardware for example when a high-energy particle strikes a transistor, resulting in a spontaneous bit-flip. Such events have been acknowledged as the source of major crashes in server systems [6]. The trend towards lower threshold voltages and tighter noise margins means that susceptibility to transient faults is increasing.

From a security perspective, transient faults (henceforth we will say simply faults) are a known attack vector. For instance, in [7,3,20] a single bit flip, regardless of how is triggered, can compromise the value of a secret key in both public key and authentication systems. In [17] it is shown how a fault (induced by holding a light-bulb near the processor!) triggers a single bit flip in a malicious but well-typed Java applet, causing it (with high probability) to do something which is otherwise impossible for well-typed bytecode: to take over the virtual machine.

Much previous work on *fault tolerance* has studied the preservation of functional behavior or mitigation of faults. For the most part techniques employ wholesale hardware replication, or at least some special-purpose hardware. For the predominantly-software-based techniques, with the exception of [24], most works do not give precise, formal guarantees.

In this work, rather than attempting to preserve full functional behavior in the presence of faults, we consider the novel problem of guaranteeing security: faults may cause a program to go wrong, but even if it goes wrong it should not leak sensitive data, no matter if the code is crafted with malicious intent (cf. [17]). The particular security characterization we study is *non-interference*, a well-established end-to-end information-flow security property which says that public outputs of a program (the *low* security channel) do not reveal anything about its secrets (the *high* security inputs).

J. Jürjens, F. Piessens, and N. Bielova (Eds.): ESSoS 2014, LNCS 8364, pp. 60–76, 2014.

Our approach has two distinguishing features. Firstly, it does not rely on special purpose hardware features (in contrast to [24]), and secondly, it makes its assumptions precise and provides formal guarantees. This latter point distinguishes our approach from software-based techniques used in the large majority of works in fault tolerance which are usually evaluated empirically, often using simulated errors. It should be noted, of course, that our goal is simply to preserve non-interference, and not to detect errors or recover from them.

In the remainder of this section we give an overview of the approach taken in this work to achieve what we called *fault-tolerant non-interference*, and summarize the main results.

The Target System and the Faulty Environment. Transient faults are a feature of hardware, so it makes sense to have an explicit hardware representation. In this paper we consider a single core machine that executes a small set of RISC-like instructions. The machine has registers and two separate memories for code and for data (§ 2.1). We assume the code memory is read-only (ROM), therefore fault-free. This is a standard assumption since memory with error correcting codes is both efficient and commonplace. On the other hand we assume that both registers and data memory are *not* fault-free. This means, in particular, that even the program-counter and hence the control flow can be affected by faults, an assumption in line with most CPU implementations. This is the feature of the system (and systems in general) which makes the problem particularly challenging.

Since we aim for precise guarantees, we assume there is no operating system between programs and the underlying hardware. This choice simplifies the implementation of our method and the security argument. In fact, since the execution of the operating system would be subject to faults, none of its abstractions could be used in a reliable way, and the code would introduce further vulnerabilities.

We assume that the fault environment can simultaneously induce multiple bit-flips in any register or any part of the data memory.

Enforcing Non-interference in the Presence of Transient Faults. Our method enforces security via program transformation. Security is defined in terms of two secrecy levels, low for public and $high$ for confidential data; low input data may influence the high outputs, but high inputs should not affect the low outputs of the system.

Our transformation combines *Secure Multi-Execution* (SME) [15] [1] with a technique known from Software-based Fault Isolation (SFI) [31] to guarantee that the security property enforced by SME is not compromised by faults.

Consider the system consisting of high and low inputs and outputs represented in Figure 1. The SME version of this system is given in Figure 2. SME deploys two isolated copies of the system, one with responsibility for computing the low outputs, and one with the responsibility of computing the high ones. In our instantiation of this idea, the “system” will be the program to be secured.

A natural approach to implementing SME is to use fair concurrency to compute independently each copy of the system. In our case, the approach has necessarily to

[1] Related ideas have appeared elsewhere [27,9,12,5]

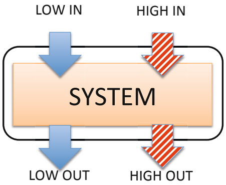

Fig. 1. Original System

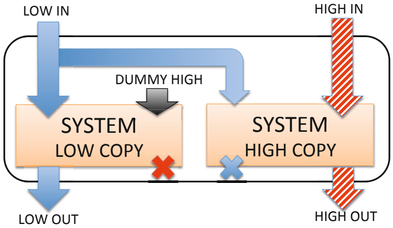

Fig. 2. Secure Multi-Execution

be more straightforward, since software and hardware supports for concurrency are missing. For this reason, SME is implemented by executing the high copy sequentially after the low one. This mandatory choice makes SME vulnerable to leakage in the presence of faults (§ 2.2-2.3). In particular:

- during execution of the low copy, a fault in the value of a pointer stored in a register could cause the high data to be loaded instead of low;
- during the execution of the high copy, a fault in the program counter can cause the control-flow to transfer to the low copy, but in a state where the registers might contain arbitrary high data.

In both of these scenarios, the low copy of the code gains access to the high data. The attacker's ability to take advantage of this may depend on the structure of the code, or the attacker's ability to recognize a leaked secret independently of the code. Nevertheless, to construct a general security mechanism based on SME, we must protect against the situations enumerated above.

A typical assumption in the analysis of fault tolerance mechanisms is the occurrence of a single fault. Similarly, we strengthen SME so that it can cope with at most some small fixed number of faults (§ 3.3). The key to preserving the strong isolation provided by SME, in the presence of up to F faults, is to

- (§3.1) separate the address space of the high and low variants of the code, and the data memory addresses over which they operate so that the addresses of the respective parts have a hamming distance[2] greater than F
- (§3.2) add address masking code, in the style of SFI, around load and jump instructions to mask the address value so that it is forced within in a safe range.

As for the original SME, our method guarantees isolation between *low* and *high* components in a language-independent manner, since systems are treated as black boxes; moreover, such isolation remains unaltered even if F faults occur during the execution. Our method guarantees *transparency* as well: if the original system had no information leaks between high inputs and low outputs, and no faults occur in the execution, then the modified system will produce the same values on the low and high channels as the original system (since the dummy high input will have no influence on the computation).

Results. For security, we formalize the semantics of the machine (§ 4.1) and precisely specify our assumptions about which faults can occur (§ 4.2). From this we formulate

[2] The number of positions for which corresponding bits of two equally sized binary words differ.

a suitable notion of non-interference (§ 4.3), where we tackle the problem that faults, when modeled as nondeterminism, can mask information flows.

Surprisingly, security is established with no semantic assumptions about the code itself. In order to guarantee transparency we need "reasonable" semantic invariants (§ 5) on memory utilization and control flow modifications performed by the source program.

2 Transient Fault Based Attacks on SME

This section illustrates the syntax of assembly programs and the inadequacy of a naive SME implementation in the presence of faults.

2.1 Syntax

Data manipulated by assembly programs are in the set Val, which is defined as the disjoint union of $\mathbb{W} \cup Ptr \cup Lab \cup DReg$. The set $\mathbb{W}$ corresponds to numeric constants, defined as machine words of n bits. Pointers to data memory, from the set $Ptr \stackrel{\text{def}}{=} \{\mathrm{ptr}\, v \mid v \in \mathbb{W}\}$, are defined as tagged machine words to keep them separated from elements in $\mathbb{W}$. We assume an infinite set of labels Lab, representing targets of jump instructions, and a finite set of general purpose registers $DReg$.

$$
\begin{array}{lllllll}
I & ::= & [l :]B \text{ such that } l \in Lab & & & & \\
B & ::= & \mathsf{load}\; r\; v & \mid \mathsf{store}\; v\; r & \mid \mathsf{jmp}\; v & \mid \mathsf{jnz}\; v\; r & \mid \\
 & & \mathsf{nop} & \mid \mathsf{move}\; r\; v & \mid BinOp\; r\; v & \mid \mathsf{out}\; ch\; r & \\
BinOp & ::= & \mathsf{add} & \mid \mathsf{or} & & & \\
P & ::= & \epsilon \mid I :: P & & & &
\end{array}
$$

Fig. 3. Assembly programs syntax

Figure 3 shows the syntax for assembly programs. We consider that every instruction I could be optionally labeled. Instruction load r v accesses the data memory and writes the value pointed by v into register r. The corresponding store v r instruction writes the content of r into the data memory address v. Instruction jmp v causes the control-flow to transfer to the instruction labeled as v. Instruction jnz v r performs the jump only if the content of register r is nonzero. Instruction move r v copies the value v into register r. $BinOp$ stands for a family of binary operators that combine values in r and v and store the result in r. A minimal such family contains an or instruction and an add instruction. The or instruction performs the logic or operation between constants in r and v; the add instruction adds the unsigned constant v to the value contained in register r, which can either be a constant or a memory pointer. All instructions presented so far are either indirect, when v is in $DReg$, or direct when v is in $Val \setminus DReg$. Instruction nop performs no computation. Instruction out ch r outputs the constant contained in r into the channel ch. Output channels are in the set $Out = \{low, high\}$.

Programs are defined as lists of instructions P. We denote the set of labels contained in a program as $lab(P)$. We require programs to be well-formed, namely not having two instruction bodies labeled in the same way. Given two programs P and P', we define program composition $P \mathbin{+\!\!+} P'$ as list concatenation, provided that $lab(P) \cap lab(P') = \{\,\}$.

2.2 Direct Control Flow and Memory Faults

We describe how faults can induce secret leakages in SME-programs. Consider Figure 4, in which an assembly program and the memory M on which it is executed are presented. Observe that M contains both a public value pub and a secret sec. The program P is intuitively secure. The first move instruction writes the memory pointer pub_p to register r_1. Then the public value pub is loaded in r_2, and sec_p overwrites pub_p in r_1. Finally, pub is output on the low channel via the last out instruction.

Since program P is secure, its SME version, written $sme(P)$, is also secure [15]. Figure 5 shows the code of $sme(P)$ and the corresponding memory. The transformed program consists of the two copies of program P, named P_{low} and P_{high}, responsible for computing public and secret values, respectively. The memory is divided into the segments μ_{low} and μ_{high} in such a way that the code in P_{low} only refers to μ_{low} and the code in P_{high} only to μ_{high}. The segment μ_{low} contains the dummy value zero ($sec'_p \mapsto 0$) instead of the secret value sec, while instructions for public outputs are replaced by nop in P_{high}. Clearly, $sme(P)$ preserves confidentiality.

P
move r_1 (ptr pub_p)
load r_2 r_1
move r_1 (ptr sec_p)
nop
out low r_2

M
$sec_p \mapsto sec$
$pub_p \mapsto pub$

Fig. 4. Secure program

We proceed to describe how a single bit flip is enough to jeopardize the security guarantees of $sme(P)$. In a machine execution, it could be possible for sec_p and pub'_p to be located at the memory addresses 000 and 100, respectively. It is then possible for pub'_p to be converted to sec_p by a single bit flip. As a consequence, the secret value sec could be loaded into r_2 by the second instruction in P_{low}, which in turn would send it on a low channel.

Bit flips in the program counter are problematic as well. Suppose the execution goes through P_{low} and completes the first nop in P_{high} without faults. At this point, the program counter contains the value 9 (1001 in binary), i.e., it points to the last instruction of P_{high}, and the register r_1 contains the pointer sec_p. However, just before the last instruction of P_{high} is executed, a bit flip in the first bit of the program counter can move the execution back to 0001, i.e., the second instruction of P_{low}. Since this occurs while r_1 contains sec_p, it is possible for P_{low} to have access to sec, and leak it on the low channel.

	$sme(P)$
P_{low}	move r_1 (ptr pub'_p)
	load r_2 r_1
	move r_1 (ptr sec'_p)
	nop
	out low r_2
P_{high}	move r_1 (ptr pub_p)
	load r_2 r_1
	move r_1 (ptr sec_p)
	nop
	nop

$sme(M)$	
$sec_p \mapsto sec$	μ_{high}
$pub_p \mapsto pub$	
$sec'_p \mapsto 0$	μ_{low}
$pub'_p \mapsto pub$	

Fig. 5. $sme(P)$ and $sme(M)$

The scenarios described above suggest that in order to guarantee security in a faulty context, SME has to separate P_{low}, P_{high}, μ_{low}, and μ_{high} in a way that tolerates bit flips in memory pointers or in the program counter, as discussed in Section 3.1.

2.3 Indirect Control Flow and Memory Faults

Faults can induce arbitrary computations *within* P_{low} and P_{high}. Although we do not attempt to preserve functional correctness in the presence of faults, performing arbitrary computations in a SME scenario has important security implications.

Consider the fragment of low code in Figure 6. Alterations in the program counter could bypass the initialization of r_1 to ptr pub_p and use an arbitrary value • as memory pointer. Hence, regardless how μ_{low} and μ_{high} are spread out in memory, it would be still possible for a pointer in P_{low} to refer to values in μ_{high}. This situation can clearly jeopardize the security guarantees of SME. Observe that arbitrary computations on P_{high}'s memory pointers do not present any security risks. After all, it is secure for P_{high} to access μ_{low}. However, perturbations in P_{high}'s control flow impose other danger.

move r_1 •

move r_1 $(\text{ptr}\, pub_p)$

nop

load r_2 r_1

Fig. 6. low code

When P_{high} is executed, faults in the program counter could induce arbitrary values to be used as jump targets. When this is the case, the control flow can be moved from P_{high} back to P_{low}, regardless how P_{low} and P_{high} are located in memory. Since secret data is often loaded into registers by P_{high}, this type of jumps presents a security risk. Observe that there is no risk for arbitrary computations to trigger jumps from P_{low} to P_{high}.

In Section 3.2 we propose to use instrumentations for instructions load, jmp, and jnz so that leaks can be prevented even in the presence of arbitrary computations.

3 Fault-Tolerant Secure Multi-execution

We present a version of SME capable of preserving confidentiality of high inputs even in a faulty environment. Our technique relies on spreading out code (P_{low} and P_{high}) and memory (μ_{low} and μ_{high}) as well as instrumenting instructions related to memory access and jumps.

3.1 Fault-Tolerant Layout for Code and Memory

Fault tolerance always involves some kind of redundancy. In our case we will use the first $F + 1$ bits of every n-bit address exclusively for keeping the hamming distance between P_{low} and P_{high}, and between μ_{low} and μ_{high}, to at least $F + 1$.

Let $distance(u, v)$ be the hamming distance between two words u and v. We will say that two words are F-separate whenever their hamming distance is greater than F.

We will work with programs for which both their size, and their run-time memory footprint, is roughly in the range $[0, 2^{n-(F+1)} - 1]$ (the exact range may be slightly smaller than this and can be calculated after some additional instructions have been

iloadSec
load r' v ↦ move r_{sp} $mask$
or r_{sp} v
load r' r_{sp}

Fig. 7. Securing load

ijmpSec
jmp v ↦ move r_{sp} $mask$
or r_{sp} v
jmp r_{sp}

Fig. 8. Securing jmp

ijnzSec
jnz v r' ↦ move r_{sp} $mask$
or r_{sp} v
jnz r_{sp} r'

Fig. 9. Securing jnz

inserted into the code according to the transformation described in the next subsection). The remaining bits of the address spaces (code and data memory) are reserved for our fault tolerance mechanism.

Let $mask$ denote the word with $F + 1$ leading 1s followed by $n - (F + 1)$ zeros.

The idea is that any address in the range $[b, t]$ (where $b < t < 2^{n-(F+1)}$) is F-separate from any address in the range $[b + mask, t + mask]$.

If μ_{high} occupies the memory addresses in the interval $[0, t]$ then we ensure that μ_{low} uses the range $[mask, t + mask]$. This clearly gives F-separation between μ_{low} and μ_{high} and thus avoids leaks due to faults in pointers handled by P_{low} (see Section 2.2).

For achieving a similar separation between P_{high} from P_{low} we add some code padding between the two copies of P such that the first instruction of P_{high} is at the ROM address $mask$. This guarantees F-separation between the addresses of instructions in P_{low} and P_{high} and thereby avoids leak due to direct faults in the program counter while executing P_{high} (see Section 2.2).

3.2 Control Flow Integrity

Faults can break the control-flow integrity of the program, causing it, for example, to jump to an arbitrary address. The two problematic instances of this problem are when (i) P_{low} loads from an address in μ_{high}, and (ii) when the destination of a jump in P_{high} points to P_{low}. We mitigate these cases using a technique which turns out to be very similar to the sandboxing approach in software-based fault isolation [31]: we mask the addresses so that they are always within a safe range. This is achieved in case (i) by transforming load instructions, and in case (ii) by transforming jmp and jnz instructions, as shown in Figures 7 to 9.

Note that for this to work we need one spare general purpose register r_{sp} – i.e., one which is not used by the original program P.

3.3 Formal Definition of Fault-Tolerant SME

Figure 10 summarizes the process of generating our fault-tolerant version of SME as a program transformation. SME reworks an assembly program P into two secure variants P_{low} and P_{high}. This requires modifications to the internal behavior of program P. The transformation consists of several steps. To obtain P_{high} from P, we first replace the instructions to write data into public channels by nops. This is done by the function o_{low}, which generates an intermediate result P'_{high}. Function jnzSec ∘ jmpSec (the symbol ∘ denotes function composition) instruments jmp and jnz instructions by applying functions in Figures 8 and 9 to the entire program.

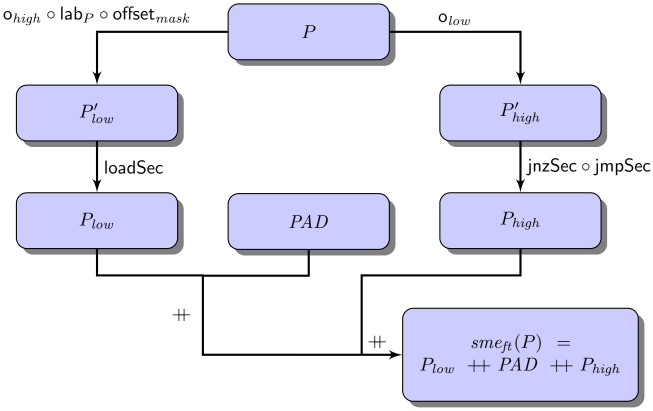
$\mathsf{o}_{high} \circ \mathsf{lab}_P \circ \mathsf{offset}_{mask}$
P
o_{low}
P'_{low}
P'_{high}
loadSec
jnzSec ∘ jmpSec
P_{low}
PAD
P_{high}
++
++
$sme_{ft}(P) = P_{low} ++ PAD ++ P_{high}$

Fig. 10. Fault-tolerant SME code transformation (sme_{ft})

Obtaining P_{low} is a bit more involved. It requires offsetting every pointer appearing in P by $mask$ so that P_{low} refers to μ_{low} (function offset_{mask}). Additionally, the transformation renames instruction labels to avoid name clashes with P_{high} (function lab_P), as well as suppressing instructions performing outputs in high channels (function o_{high}).

The instrumentation of load is done by function $\mathsf{loadSec}$ (based on the auxiliary function in Figure 7), thus finally obtaining P_{low}. Once P_{low} and P_{high} are obtained, in order for F-separation to hold between them, the transformation adds some padding code, named PAD. All instructions in PAD are jumps to the first instruction of P_{high}, and the length of PAD guarantees the first instruction of P_{high} is located at the address $mask$ (recall Section 3.1).

Initial memory configuration. Consider the initial memory M for P in Figure 11. We assume that the program uses the memory interval $\mu = [0, t]$, where the first s words in M are secrets (labeled $high_{in}$), the subsequent words are public values (low_{in}) and the rest is uninitialized (in white). We require s to be within the range $[0, 2^{n-(F+1)} - 1]$ to ensure the separation between μ_{high} and μ_{low} is possible (Section 3.1).

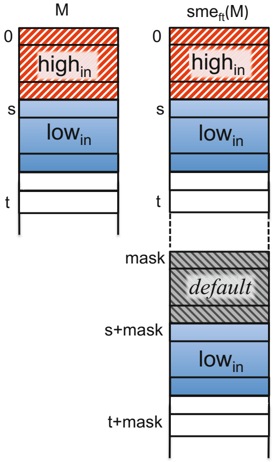

Fig. 11. Initial memory M and transformed version $sme_{ft}(M)$

We also require that M only contains values from $\mathbb{W}$. The security of the method does not depend on this assumption, but for the transformation to preserve the non-faulty behavior of secure runs of the program we will need such requirement on input. We return to this issue in Section 5. Under these assumptions, the initial memory for $sme_{ft}(P)$, which we denote by $sme_{ft}(M)$, corresponds to the right side of Figure 11. Notice that μ_{high}, the portion of the memory to be used by P_{high}, is the same as μ, whereas P_{low} will use μ_{low} which is located in the memory interval $[mask, t + mask]$. In μ_{low} the words representing the secret are initialized to a default value (marked "default" in the figure). For the sake of simplicity, we do not require $sme_{ft}(P)$ to take care of memory rearrangement itself – we assume the preparation of $sme_{ft}(M)$ is external to SME. We assume initial registers to be all uninitialized for P, therefore they will be uninitialized for $sme_{ft}(P)$ as well.

Optimizing sme_{ft}. It might appear redundant to modify memory pointers in P_{low} *and* instrument direct load instructions according to Figure 7 (and similarly for control flow labels in P_{high} and functions in Figures 8 and 9). For many sensible programs this is indeed the case, such as the *safe* programs characterised in § 5.

Redefining $mask$. Recall that in Section 3.1 we define $mask$ as the mask used to obtain F-separation of memory and code. When it comes to the code, we assume that the size of P_{low} is the same as P_{high}. However, this assumption is no longer true for P_{low} and P_{high} produced by sme_{ft} due to the instrumentations of load, jmp and jnz instructions. This is not a major problem. It is enough to pad with nops P_{low} or P_{high} to match their sizes. For simplicity, we omit this step in our schematic description.

4 Security Guarantees Provided by sme_{ft}

In this section we state the security property bestowed by sme_{ft} on transformed programs. To do this we define a formal semantics for the RISC machine; extend it to model faults; define non-interference for faulty runs; state the security theorem: any program transformed by sme_{ft} corresponds to a machine program which is non-interfering for runs with no more than F faults. For space reasons most of the details are not given here; we refer to the full version [13].

4.1 Semantics

To give a precise semantics to faults we need to work at the level of concrete programs, i.e., *machine code*, which are lists of concrete instructions. Compared to assembly instructions from Figure 3, concrete instructions are not labeled, and their arguments are register names or machine words. This formalization of machine code is sufficiently concrete to describe the class of faults we wish to model. In particular, a concrete encoding of the register names is not made explicit because we do not consider faults in the code memory, and because registers are not addressable indirectly. We sometimes write $P(i)$ to denote the ith concrete instruction in the instruction list P.

$$\textbf{DLoad}\ \frac{P(pc) = \mathsf{load}_d\ r\ w}{\langle P, Reg, M\rangle \xrightarrow{\tau} \langle P, Reg^{+}[r \mapsto M(w)], M\rangle}$$

$$\textbf{DAdd}\ \frac{P(pc) = \mathsf{add}_d\ r\ w \quad Reg(r) + w = w'}{\langle P, Reg, M\rangle \xrightarrow{\tau} \langle P, Reg^{+}[r \mapsto w'], M\rangle}$$

$$\textbf{DJnz-A}\ \frac{P(pc) = \mathsf{jnz}_d\ w\ r \quad Reg(r) \neq 0}{\langle P, Reg, M\rangle \xrightarrow{\tau} \langle P, Reg[pc \mapsto w], M\rangle}$$

$$\textbf{Out}\ \frac{P(pc) = \mathsf{out}\ ch\ r}{\langle P, Reg, M\rangle \xrightarrow{ch!Reg(r)} \langle P, Reg^{+}, M\rangle}$$

Fig. 12. Concrete Semantics (selected rules)

Most assembly instructions have two explicit versions in the concrete domain: a *direct* version, such as $\mathsf{load}_d\ r\ w$ which loads the value contained at memory address w into the register r, and an *indirect* version, such as $\mathsf{load}_i\ r\ r'$ which fetches the memory address of the data to be loaded from register r'. There are two exceptions to this: the nop instruction, which does not require any parameter, and the out instruction, which has no direct formulation. Observe that, similarly to register names, channel names are not encoded.

Assembly programs are converted to concrete ones by the function loader. The function converts abstract values Val into machine words. In particular this amounts to stripping the pointer tag away from the pointers, and resolving code labels to ROM addresses. The function loader is also responsible for mapping all abstract instructions into their direct or indirect versions. The details are straightforward and not presented here [13].

Configurations of the concrete machine are given by a triple $\langle P, \mathit{Reg}, M \rangle$, where P is the concrete program, $\mathit{Reg} \in \mathit{DReg} \cup \{pc\} \to \mathbb{W}$ is the *(Concrete) Register Bank* and $M \in \mathbb{W} \to \mathbb{W}$ is the *(Concrete) Data Memory*.

The fault-free semantics of concrete programs is given as a labeled transition system. The labels on transitions indicate the observable output of each clocked machine step, and are either τ, a label marking just the passage of time, or an output label, indicating a word output on a specific channel. All labels are in $\mathit{Act} = \{\mathit{low}!w | w \in \mathbb{W}\} \cup \{\mathit{high}!w | w \in \mathbb{W}\} \cup \{\tau\}$. A representative selection of reduction rules for the concrete machine are presented in Figure 12. We use Reg^{+} as a shorthand for $\mathit{Reg}[pc \mapsto \mathit{Reg}(pc) + 1]$ and we abbreviate $P(\mathit{Reg}(pc))$ as $P(pc)$. Modelling instructions as consecutive words implies that it is impossible to jump to an address which is not aligned with the beginning of an instruction; this assumption corresponds to the implementation of simpler RISC architectures such as ARM versions 1 and 2.

4.2 Modeling Faults

Our aim will be to describe the overall behavior of a fault-prone system as simply as we can, while still permitting reasoning about non-interference. The core idea is to model the transitions of the system in the presence of faults with a labeled transition system obtained by interleaving the machine transitions with a nondeterministic flipping of zero or more bits. As described previously, the fault-prone bits of the machine are any of the register bits, and any bits in the data memory.

We need some notation to talk about bit flips. Recall machine words are n bits long. Let us define the set of *locations* at which a fault may occur as:

$$\mathit{Loc} \overset{\text{def}}{=} \{(r, i) \mid r \in \mathit{DReg} \cup \{pc\}, i \in \{1, \ldots, n\}\} \cup \{(k, i) \mid k \in \mathbb{W}, i \in \{1, \ldots, n\}\}$$

For a machine configuration C and location $l \in \mathit{Loc}$ we will write $C[l]$ to denote the value of the bit specified by l in C; for any $b \in \{0, 1\}$ we write $C[l \mapsto b]$ to denote the configuration obtained from C by updating the location l to b.

Let L range over the (possibly empty) subsets of locations. We express bit flips in the values of a given subset L of locations by using the function flip defined as $\mathsf{flip}(C, L) = C[l \mapsto \neg\, C[l], l \in L]$, which flips every bit of locations L in the machine configuration C.

$$\frac{\mathsf{flip}(C, L) \xrightarrow{a} C' \quad L \subseteq \mathit{Loc}}{C \overset{a}{\rightsquigarrow} C'}$$

We can now define faulty systems with labeled transitions ($\overset{a}{\rightsquigarrow}$, $a \in \mathit{Act}$) with the transition rule to the right. It can be seen from the rule that our fault model assumes that the transitions of the system are instantaneous (a common assumption, but a potential source of inaccuracy – a point we return to in the conclusions). The fact that faults can occur between transitions is modeled by allowing any fault to occur before any transition of the system is taken. The number of faults occurring in a given transition is $|L|$, and is not constrained in this rule, but will be constrained at the level of *runs*.

4.3 Fault-Tolerant Non-interference

This section formalizes the confidentiality guarantees of our approach in the presence of faults.

Since the faulty system is nondeterministic, one might consider a simple *possibilistic* notion of non-interference — secret values should not influence the *set of possible public outputs* of the faulty system. This notion is not adequate because unfortunately errors might occur anywhere, in particular on public values, therefore any program is capable to produce any possible output!

This is an instance of a known weakness of possibilistic non-interference [18,22]. A standard fix is to adopt a *probabilistic* notion of non-interference – the probability distribution of public outputs is unaffected by the secrets in the presence of errors – assuming an attacker can perform probability measures. In this paper, however, we adopt a different approach: we permit the attacker to observe *exactly when and where faults occur* in a given run, along with output events in the low channel and the passage of time. This model leads to a security definition which seems stronger than the probabilistic one, but in fact we have shown [14] that the two notions are equivalent for the computational model considered here.

We start concretising the attacker's view of a system by defining function $low \in Act \rightarrow \{low!w | w \in \mathbb{W}\} \cup \{\tau\}$. More precisely, $low(a)$ returns a if $a = low!w$, and returns τ otherwise. Now we can define the semantics of the faulty system from the attacker's perspective as a labeled transition system given by the following transition rules:

$$\textbf{Step}\ \frac{\mathsf{flip}(C,L) \xrightarrow{a} C'}{C \overset{L,low(a)}{\rightsquigarrow} C'} \qquad \textbf{Stuck-1}\ \frac{\mathsf{flip}(C,L) \not\rightarrow}{C \overset{L,\tau}{\rightsquigarrow} \mathsf{flip}(C,L)} \qquad \textbf{Stuck-2}\ \frac{C \not\rightarrow}{C \overset{L,\tau}{\rightsquigarrow} C}$$

The attacker observations imply that termination of the system is not directly observable and that once a system reaches a stuck configuration, faults have no further effect.

We can now state our security condition. We say a machine configuration is *initial* if (i) $Reg(pc) = 0$, (ii) $Reg(r_{sp}) = 2^n - 1$ (so it never points to low code/high data), and (iii) secrets are stored in the first s words of the memory (Figure 11).

We say two initial configurations C and C' are *low equivalent*, written as $C =_{low} C'$ if they differ, at most, on the first s words of the heap.

We say that a sequence $\sigma = L_0, a_0, \ldots L_{n-1}, a_{n-1}$ is a *low run* of a system state C_0 whenever there exist states $C_1, \ldots, C_n$ such that $C_i \overset{L_i,a_i}{\rightsquigarrow} C_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. The number of faults exhibited by σ is $\Sigma_{i=0}^{n-1} |L_i|$.

Definition 1 **(F-Fault-Tolerant Non-interference).** *An initial configuration C is F-fault-tolerant non-interfering if for all initial configurations C' such that $C =_{low} C'$, the set of low runs exhibiting no more than F faults are the same for C and C'.*

We say that an assembly program P is F-fault-tolerant non-interfering if all initial configurations relative to P, namely $\langle \mathsf{loader}(P), Reg, M\rangle$ are F-fault-tolerant non-interfering.

Theorem 1 **(Non-interference induced by sme_{ft}).** *If $sme_{ft}(P) = P'$ then P' is F-Fault-tolerant non-interfering.*

The theorem is proved by showing that (i) all memory accesses in P_{low} are performed towards addresses that are F-separate from μ_{high} and (ii) once the computation reaches P_{high} it cannot be moved back to P_{low}.

Both properties depends on the layout of code and data memory, together with on the invariant property on r_{sp}. In particular we can show that in the absence of faults, the value contained in r_{sp} is in the range $[mask, 2^n - 1]$, whereas in the presence of faults the content of r_{sp} is never in the range $[0, 2^{n-(F+1)} - 1]$. For a detailed proof refer to [13].

Definition 1 is both termination and (logical) timing sensitive: we require that any two runs of the system (that exhibit at most F faults) correspond to the same sequence of observable events, regardless of secret data. Not only output values must be the same, but the instant in which they occur must coincide as well. Hence, Theorem 1 guarantees that our transformation technique can secure all programs whose timing and termination behavior can induce leaks.

5 Transparency Guarantees Provided by sme_{ft}

We have shown that the transformed programs meet the goal of non-interference in the presence of faults. We have done so with no semantic assumptions about the code itself. The only *syntactic* assumptions are on the size of the code, which is required to be small enough to accommodate the transformation in the ROM, on the amount of secret data in the initial memory, and on the registers utilization – we require at least one spare register.

Does the transformation sme_{ft} preserve the behavior of programs? The answer, in general, is no. Firstly, programs which are intrinsically insecure exhibit a different behavior under standard SME. This alteration in the semantics is done in order to enforce confidentiality. It could be said that “software faults”, i.e., instructions leaking secret data, are being mitigated by SME. However, even when the original program is secure, our transformation modifies the size and layout of the original program and the absolute location of data in memory. In general machine code programs can be sensitive to such transformation, and behave in an arbitrarily different way.

For this reason, transparency guarantees can be given only for programs which are “sensible” and secure for fault-free runs. We consider a program “sensible” when it is *safe* and *bounded.* A program is *safe* when, roughly speaking, it is not sensitive to the absolute addresses of its instructions in the ROM, or the absolute addresses of the memory that it accesses. A program is *bounded* when there is a known upper bound on the region of memory that it will address.

For any "sensible" program, the following theorem holds:

Theorem 2 (Transparency). *(informal statement) Let P be a non-interfering, "sensible" assembly program. If the low copy P_{low} always terminates, then the SME transformed program $sme_{ft}(P)$ yields the same sequence of values on each of the respective output channels as P for any fault-free run.*

A formal account of Theorem 2 (and its proof) can be found in the full version of the paper [13].

In this work the characterization of safe and bounded programs is obtained via an abstract machine for the language. The abstract machine characterises those programs which never exhibit certain "bad" behaviours. This is in the same spirit as e.g. Leroy's compiler correctness proof [21]. We expect that any program correctly compiled from a strongly-typed high level language, and which has a statically known memory footprint, will be a safe and bounded program. To give these guarantees formally one could use a verified compiler, or it could be achieved by compiling to a typed version of our assembly language (see, for example, [23]) which ensures that the produced code is safe and bounded. However, these endeavours lie outside the scope of the present paper.

Notice that for Theorem 2 to hold we require the low copy of the source program to terminate on all input. This means that, in general, transparency does not hold for programs that are nonterminating by construction (e.g. server applications). However, this does not compromise security: Theorem 1 holds for this class of programs as well.

6 Related Work

Language Based Dependability. The use of application-layer techniques for achieving fault tolerance have been widely studied. De Florio and Blondia survey the field [16] and classify the various ways in which fault tolerance can be added, and what kind of faults are supported. Notably, none of the techniques surveyed at that time either deal with tolerance with respect to security properties, or with techniques that give precise semantic guarantees.

More recently, Project Zap [1] has applied language based techniques to transient faults modeling and analysis with the goal of providing formally verifiable dependability methods. The closest to our work in the Zap series is the work on fault-tolerant typed assembly language of Perry et al [24]. We use an abstract machine to characterize the class of programs for which our method is applicable. Our characterization is more liberal than a typical typed assembly language, but a typed assembly language could nevertheless be used as a sound method to prove that a program is safe and bounded. Both in that work and in ours, transient faults have a semantic interpretation as nondeterministic transitions that can happen at anytime and anywhere in the faulty hardware. Since we do not aim at functional correctness preservation, we can be more liberal in the class of faults we admit (more than one bit flipped at a time) and in the hardware components the concrete machine operates on. In [25] the attention is solely focused on detecting control flow modifications induced by transient faults. The method, unlike

[24], is purely software based. However, detectability is possible only for programs that obey a strict control-flow discipline, and under the assumption that at most a single bit flip occurs. Once again, our ability to cope with a bigger class of control flow errors comes from the fact that we aim for a weaker property; arbitrary control flow alterations inside P_{low} or P_{high} executions do not pose security threats.

Fault Isolation Techniques. As mentioned previously, the techniques we use to mask addresses to prevent dangerous loads and jumps can be found in the software-based techniques for fault isolation (SFI) introduced by Wahbe *et al* [31] for sandboxing untrusted code. A similar address-masking technique is used in [10] for mitigating the effects of transient faults. Also, principles from SFI are also implemented in [2], where the authors define a method to prevent an active attacker from corrupting the control flow integrity of a program.

It should be noted, however, that the "faults" targeted by SFI are those caused by buggy/malicious code or data. The SFI techniques, in isolation, are able to protect from the effects of some but not all of the transient faults studied here.

What we said for software based methods also hold for sandboxing techniques using special operating system or hardware features – they are not designed for and do not protect against all transient faults, and may increase the attack surface (via increased code or by relying on special purpose registers).

Fault Tolerance vs Non-Interference. As we have shown in our result, fault tolerance and non-interference present interesting connections, and we believe that our combination is a novel one. However other connections between the two concepts have been noted in a number of other works.

The *Strong Security* notion introduced by Sabelfeld and Sands in [29] for multi-threaded programs is shown to be strong enough to guarantee an unrestricted form of fault-tolerant non-interference in [14], providing a more restrictive class of transient faults are considered (faults cannot corrupt the control flow integrity). In a similar way, programs that are secure according to the definition in [28], an extension of [29] to distributed systems, can be shown to retain security regardless of faults occurring in network communications. It is not surprising that both cases cannot cope against faults in the control flow since, as we have shown in Section 2, control flow alterations introduce completely unexpected information flows.

Another interesting aspects of the comparison between fault tolerance and non-interference was observed by Weber [33]. In this work the author explores a non-interference-like characterisation of fault tolerance in terms of program semantics. A more general view on the connection between enforcement mechanisms for information flow properties and dependability goals is proposed by Rushby [26]. Overall the techniques used in the present work can be understood in terms of the general partitioning mechanisms described by Rushby. In particular what Rushby calls *spatial partitioning* corresponds to our separation of memory addresses (albeit within the same physical memory); *temporal partitioning* characterises what we achieve by ensuring that low events happen before high events, since this ensures that the timing of high events cannot influence low events.

Security Preservation in the Presence of Transient Faults. Our method guarantees that security of programs, expressed in terms of F-Fault-Tolerant Non-interference, is preserved even when a limited number of bit flips occur. Other forms of security preservation in faulty environments have been studied, particularly in cryptography.

In [4] authors illustrate several transient-fault based attacks on RSA and Discrete Logarithms cryptographic schemes, together with software countermeasures. Such protection mechanisms involve either some form of replication (they basically require to repeat the computation twice and check the result for fault detection) or a more intensive usage of randomness in the intermediate stages of cryptographic operations to increase the unpredictability of the result.

In [11] authors show how the parameters of an elliptic curve cryptosystem can be compromised by transient faults, and illustrate how a comparison mechanism is sufficient to prevent the attack from being successful. In particular the method compares the working copies of said parameters (located in a faulty hardware component) to their original counterparts (stored in fault-free hardware) in several stages of the computation. Canetti et al [8] discuss security in the presence of transient faults for cryptographic protocol implementations where they focus on how random number generation is used in the code. Harrison et al consider [19] a “confinement problem in the presence of faults”, but their work concerns faults in the sense of abnormal termination of software, and the proper confinement thereof.

7 Conclusion and Further Work

We have presented a technique to make programs secure despite a small number of faults, and characterized when the method preserves the behavior of programs. The problem we study is itself novel, and relative to the faults we model, it is notable that our technique does not demand special hardware, and is capable of tolerating multi-bit errors.

Perhaps the main weakness of the present work is the fault model itself. While we model faults in all the main state elements of the machine, we do not model faults in lower-level structures, such as pipelines or in the combinatorial circuits. This shortcoming seems to be shared with much work on fault tolerance (although we do, at least, model faults in the program counter) – in particular works which focus on fault injection e.g. [30]. One might speculate that many faults occurring at the lower level of abstraction are adequately modeled by flipping a few bits in a register, but there seems to be little work to verify this. One of them, by Wang *et al* [32], suggests that lower-level faults are notably rare.

A precise account about the efficiency of our approach is left for further work. An approximate estimation of the overhead can be determined by considering that the system is basically run twice, and all the load and jump instructions are expanded in macros of three instructions each.

Acknowledgment. Many thanks to Johan Karlsson, Ioannis Sourdis, Georgi Gaydadjiev, Arshad Jhumka and the anonymous referees for useful comments and observations. This work was partially financed by grants from the Swedish research agencies VR and SSF, and the European Commission EC FP7-ICT-STREP WebSand project.

References

1. The zap project, `http://sip.cs.princeton.edu/projects/zap/` (accessed: February 20, 2013)
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, pp. 340–353. ACM, New York (2005), `http://doi.acm.org/10.1145/1102120.1102165`
3. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault attacks on rsa with crt: Concrete results and practical countermeasures. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 260–275. Springer, Heidelberg (2003)
4. Bao, F., Deng, R., Han, Y., Jeng, A., Narasimhalu, A., Ngair, T.: Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In: Christianson, B., Crispo, B., Lomas, M., Roe, M. (eds.) Security Protocols 1997. LNCS, vol. 1361, pp. 115–124. Springer, Heidelberg (1998)
5. Barthe, G., Crespo, J.M., Devriese, D., Piessens, F., Rivas, E.: Secure multi-execution through static program transformation. In: Giese, H., Rosu, G. (eds.) FORTE/FMOODS 2012. LNCS, vol. 7273, pp. 186–202. Springer, Heidelberg (2012)
6. Baumann, R.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability 5(3), 305–316 (2005)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of Cryptology 14, 101–119 (2001)
8. Canetti, R., Herzberg, A.: Maintaining security in the presence of transient faults. In: Desmedt, Y.G. (ed.) Advances in Cryptology - CRYPTO 1994. LNCS, vol. 839, pp. 425–438. Springer, Heidelberg (1994)
9. Capizzi, R., Longo, A., Venkatakrishnan, V.N., Sistla, A.P.: Preventing information leaks through shadow executions. In: Proceedings of the 2008 Annual Computer Security Applications Conference, ACSAC 2008. IEEE Computer Society (2008)
10. Chang, J., Reis, G., August, D.: Automatic instruction-level software-only recovery. In: DSN 2006, pp. 83–92 (2006)
11. Ciet, M., Joye, M.: Elliptic curve cryptosystems in the presence of permanent and transient faults. Des. Codes Cryptography 36(1), 33–43 (2005)
12. Cristiá, M., Mata, P.: Runtime enforcement of noninterference by duplicating processes and their memories. In: WSEGI 2009, Argentina. 38 JAIIO (2009)
13. Del Tedesco, F., Russo, A., Sands, D.: Fault tolerant non-interference (extended version) (2013), `http://www.cse.chalmers.se/~tedesco/papers/essos14.pdf`
14. Del Tedesco, F., Russo, A., Sands, D.: A theory of fault tolerance noninterference (preliminary) (2013)
15. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proc. of the 2010 IEEE Symposium on Security and Privacy, SP 2010. IEEE Computer Society (2010)
16. Florio, V.D., Blondia, C.: A survey of linguistic structures for application-level fault tolerance. ACM Comput. Surv. 40(2) (2008)
17. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: SP 2003, IEEE Computer Society, Washington, DC (2003)
18. Gray, J.W., Probabilistic, I.: interference. In: Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy, pp. 170–179 (1990)
19. Harrison, W.L., Procter, A., Allwein, G.: The confinement problem in the presence of faults. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 182–197. Springer, Heidelberg (2012)

20. Kim, C., Quisquater, J.J.: Fault attacks for crt based rsa: New attacks, new results, and new countermeasures. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 215–228. Springer, Heidelberg (2007)
21. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009), http://dx.doi.org/10.1007/s10817-009-9155-4
22. McLean, J.: Security models and information flow. In: Proc. IEEE Symposium on Security and Privacy, pp. 180–187. IEEE Computer Society Press (1990)
23. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system f to typed assembly language. ACM Trans. Program. Lang. Syst. 21(3), 527–568 (1999)
24. Perry, F., Mackey, L., Reis, G.A., Ligatti, J., August, D.I., Walker, D.: Fault-tolerant typed assembly language. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 42–53. ACM, New York (2007)
25. Perry, F., Fisher, K.: Reasoning about control flow in the presence of transient faults. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 332–346. Springer, Heidelberg (2008)
26. Rushby, J.: Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (June 1999); also to be issued by the FAA
27. Russo, A., Hughes, J., Naumann, D.A., Sabelfeld, A.: Closing internal timing channels by transformation. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 120–135. Springer, Heidelberg (2008)
28. Sabelfeld, A., Mantel, H.: Static confidentiality enforcement for distributed programs. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 376–394. Springer, Heidelberg (2002)
29. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings of the 13th IEEE Workshop on Computer Security Foundations, CSFW 2000, p. 200. IEEE Computer Society, Washington, DC (2000)
30. Skarin, D., Barbosa, R., Karlsson, J.: Goofi-2: A tool for experimental dependability assessment. In: Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (2010)
31. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP 1993, pp. 203–216. ACM, New York (1993), http://doi.acm.org/10.1145/168619.168635
32. Wang, N.J., Quek, J., Rafacz, T.M., Patel, S.J.: Characterizing the effects of transient faults on a high-performance processor pipeline. In: International Conference on Dependable Systems and Networks, DSN 2004 (2004)
33. Weber, D.G.: Formal specification of fault-tolerance and its relation to computer security. In: Proceedings of the 5th International Workshop on Software Specification and Design, IWSSD 1989, pp. 273–277. ACM, New York (1989)