Fault-Resilient Non-interference

Filippo Del Tedesco
Admeta AB
Gothenburg, Sweden

David Sands, Alejandro Russo
Chalmers University of Technology
Sweden

***Abstract*—Environmental noise (e.g. heat, ionized particles, etc.) causes transient faults in hardware, which lead to corruption of stored values. Mission-critical devices require such faults to be mitigated by fault-tolerance – a combination of techniques that aim at preserving the functional behaviour of a system despite the disruptive effects of transient faults. Fault-tolerance typically has a high deployment cost – special hardware might be required to implement it – and provides weak statistical guarantees. It is also based on the assumption that faults are rare. In this paper, we consider scenarios where security, rather than functional correctness, is the main asset to be protected. Our main contribution is a theory for expressing confidentiality of data in the presence of transient faults. We show that the natural probabilistic definition of security in the presence of faults can be captured by a possibilistic definition. Furthermore, the possibilistic definition is implied by a known bisimulation-based property, called Strong Security. We illustrate the utility of these results for a simple RISC architecture for which only the code memory and program counter are assumed fault-tolerant. We present a type-directed compilation scheme that produces RISC code from a higher-level language for which Strong Security holds – i.e. well-typed programs compile to RISC code which is secure despite transient faults. In contrast with fault-tolerance solutions, our technique assumes relatively little special hardware, gives formal guarantees, and works in the presence of an active attacker who aggressively targets parts of a system and induces faults precisely.**

I. Introduction

Transient faults, or soft errors, are alterations of the state in one or more electronic components, e.g., bit flips in a memory module [24]. In some situations, we are willing to accept that a system may fail due to transient faults, for example, that an occasional message may be lost or corrupted. The problem we address in this paper is how to prevent transient faults (which will be referred to as "faults" in the rest of the paper) from compromising the security of the system – for example, by allowing an attacker to access a secret.

It has indeed been shown that bit flips can effectively be used to attack, e.g., the AES encryption algorithm and reveal the cipher key [7]. More recently, it has been shown that faults occurring in the computation of an RSA signature can permit an attacker to extract the private key from an authentication server [26]. Moreover, it is not just the security of crypto systems that is potentially vulnerable. Risks are greatly amplified if the attacker can influence the code executed by a system that can experience faults: it has been demonstrated that a single fault can cause (with high probability) a malicious but well-typed Java applet to violate the fundamental memory-safety property of the virtual machine [16].

Traditional countermeasures against faults aim to make devices fault-tolerant, i.e. able to preserve their functional behavior despite soft errors. Most fault-tolerance mechanisms are based on redundancy: resources are replicated (either in software or in hardware) so that it is possible to detect, if not repair, anomalies. Typically, such solutions are designed for protection against a very simple fault model, in which a limited number of faults are assumed. Moreover, the formal guarantees of software-based fault-tolerance mechanisms, according to Perry et al. [27], are usually not stated – it is more common that their efficacy is explored statistically. In reasoning about security it is more difficult to give a statistical model of the environment's behaviour – it might be that the environment is an active adversary with a laser and a stopwatch (cf. [31]) rather than just passive background radiation. Similarly, experimental evidence based on typical deployment of representative programs is not likely to be so useful if an attack, as it often does, requires an atypical deployment of an unusual program.

Overview of the Paper and Contributions In order to be precise about the guarantees that we provide and the assumptions we make, we develop a formal, abstract model of fault-prone systems, attackers who can induce faults based on the observations made of the system's public output, and the resulting interaction between the two (Section II-A). From this, we define our central notion of security in the presence of faults, *probabilistic fault-resilient non-interference* (PNI). A system is secure in the presence of faults if, for any attacker influencing the injection of faults, the probability of a given public output is independent of the secrets held by the system (Section II-B).

Reasoning directly about fault-resilient non-interference is difficult because (i) it demands reasoning about probabilities, and (ii) it quantifies over all possible attackers (in a given class).

Our contribution tackles these difficulties with two key theorems. **Theorem 1** (Section II-C) shows that PNI can be characterised exactly by a simpler *possibilistic* definition of noninterference, defined by assuming that the exact presence and location of faults are observable to the attacker. This removes difficulty (i), the need to reason about probabilities. **Theorem 2** (Section II-D) then shows that the possibilistic definition is implied by a form of strong bisimulation property, adapting ideas from its use in scheduler independent security of concurrent programs [30]. This theorem eliminates

difficulty (ii), enabling PNI to be proved without explicit quantification over all attackers.

Application (Section III) We illustrate the utility of Theorem 2 in a specific setting: a fault-prone machine modeling a program running on a RISC-like architecture, where faults can occur in registers and data memory, but not in the part of the memory storing the program instructions, or in the program counter register. The aim is to give a sound characterisation of RISC programs which, when placed in the context of the machine, give a system which is secure despite faults. Our approach combines ideas from security type systems for simple imperative programs with a *type-directed compilation* scheme. Roughly speaking, our type-directed compilation of simple imperative programs guarantees that a well-typed program compiles to RISC code which, for this architecture, is secure despite faults.

Our innovative perspective on the problem of guaranteeing security in the presence of transient faults combines a lot of concepts from other works, coming from both security and dependability literature. In order to clarify these connections, we discuss related works in Section IV, whereas in Section V we focus on the main limitations of our results (including those shared with other approaches). We complete the paper in Section VI, where we briefly summarize the conclusions of our study.

> *Due to space limitations, the full details of this work, and in particular all of the proofs relating to Section III, are presented in the extended version of the paper [13].*

II. A Theory for Probabilistic Fault-Resilient Non-Interference

We begin (Section II-A) by setting up a rigorous but abstract semantic model for systems that can experience transient faults, and for environments inducing these faults. Then, we establish a model for their interactions and a security definition (Section II-B) for the composition of a system with a fault environment as *probabilistic fault-resilient non-interference* (PNI). We continue (Section II-C) by simplifying the security model introduced for PNI in two steps. Firstly, we propose a simpler, nondeterministic fault model for fault-prone systems. This model leads to a more straightforward notion of security called *possibilistic fault-resilient non-interference* (PoNI)[1]which is then shown to be equivalent to PNI. Finally, we introduce (Section II-D) the notion of Strong Security (SS), a bisimulation-based security condition which is proved stronger than PoNI, hence PNI.

A. Probabilistic Fault-Resilient Non-Interference

In this work we focus on systems that can be modelled as deterministic labelled transition systems (intuitively, this corresponds to having a deterministic program running on some particular input). We assume that at any point of their execution, these systems can experience a transient fault, and for this reason we call them fault-prone systems. Since we have to reason about bit flips, we model the state of a system as just a collection of bits, each of which is identified by a unique *location*. The locations are partitioned into a *fault tolerant part* of the system (e.g. memory with error correcting codes), and a faulty part. Only the faulty locations are affected by soft errors from the environment. The labels of the transition system model outputs of the fault-prone system and "clock ticks" (τ) which mark the discrete passage of time. We assume that some outputs can be distinguished by an external observer, whereas the others appear as τ.

Definition 1 (Fault-prone System): A fault-prone system Sys is defined as $Sys = \{Loc, Act, \rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}\}$, where:

- The set Loc is finite and contains all the locations (addresses) of the system. It is partitioned into a fault-tolerant subset and a faulty one, namely T and F. Fault-tolerant locations are not affected by soft errors, whereas bits stored in the faulty part of the hardware can get flipped because of transient faults.
- Act includes system outputs and a distinguished silent action τ marking the passage of time. We assume that "public" observations (what an attacker can see) are limited to events in $LAct \subseteq Act$, whereas any other action performed by the system is observed as τ.
- The transition relation ($\rightarrow$) formalizes the system behavior. Given the set of all possible states for the system, namely $\mathcal{S}$ which are functions from Loc to $\{0, 1\}$, the transition relation $\rightarrow$ models how the system evolves. We write transitions in the usual infix manner $S \xrightarrow{a} S'$ for $S, S' \in \mathcal{S}$. The system is deterministic in the sense that for any $S \in \mathcal{S}$ there is at most one state S' and one action l and a transition $S \xrightarrow{l} S'$.

We now introduce a probabilistic fault model, which induces transient faults in F, the fault-prone subset of $\mathcal{S}$. We refer to this model as the *environment*, or, synonymously, as the *attacker*.

A simple environment can be modeled as an agent that induces bit flips in the faulty locations with some fixed (small) probability. Unfortunately, this representation does not capture many realistic scenarios: for example, some locations in the system may be more error-prone than others, and (due to high densities of transistors) the probability of a location being flipped may be higher if a neighboring bit is flipped. We could therefore consider environments that can induce faults according to a probability distribution over the powerset of faulty locations, where the probability associated with a given set of locations L is the probability that exactly the bits of L are flipped before the next computation step occurs. However, such static model of the environment is not sufficient to model an active attacker. An active attacker may have physical access to the system (e.g., a tamper-proof smart card), and may be able to influence the likelihood of an error occurring at a specific location and time with high precision (see, e.g., [31]). What is more, the attacker may do this in a way that depends on the previous observations, namely the passage of time and

[1]A similar but weaker notion of security has been presented in [12].

the publicly observable actions. We therefore formalize these capabilities of an environment as follows (recall that $\wp(A)$ here denotes the *powerset* of a given set A).

Definition 2 (Fault environment):

Consider a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$. A fault environment (Err, Fault) for Sys consists of:

- a labelled transition system $Err = \langle \mathcal{E}, LAct \cup \{\tau\}, \rightsquigarrow \subseteq \mathcal{E} \times (LAct \cup \{\tau\}) \times \mathcal{E} \rangle$, where $\mathcal{E}$ represents the set of environment's states;
- a function $\mathsf{Fault} \in \mathcal{E} \rightarrow \wp(F) \rightarrow [0,1]$ such that for all states $E \in \mathcal{E}$, we have that $\mathsf{Fault}(E)$ is a probability distribution on sets of locations.

We require that for all states $E \in \mathcal{E}$ and for all actions $a \in LAct \cup \{\tau\}$ there exists a unique state $E' \in \mathcal{E}$ such that $E \overset{a}{\rightsquigarrow} E'$. Intuitively, the state $E \in \mathcal{E}$ determines the probability that a given set of locations (and no others) will experience a fault in the current execution step of the system. At each step, the state of the attacker evolves by the observation of "low" events and the passing of time.

Example 1: We illustrate the use of Definition 2 by characterizing the simplest most uniform fault-inducing environment: a bit flip may occur in any location with a fixed probability ϵ. To model this as a fault environment, we need only a trivial transition system involving the single state $\bullet (\mathcal{E} = \{\bullet\})$, whose transitions are $\forall a \in LAct \cup \{\tau\}. \bullet \overset{a}{\rightsquigarrow} \bullet$. As for the Fault function consider a set $L \subseteq F$ where $|L| = k$ and $|F| = n$. Then the probability that flips occur in all locations in L and nowhere else is equal to the probability of faults in every location in L, namely ϵ^k, multiplied by the probability of the remaining locations *not* flipping, namely $(1-\epsilon)^{n-k}$. Hence

$$\mathsf{Fault}(\bullet)(L) = \epsilon^k (1-\epsilon)^{n-k} \quad \text{where } k = |L| \text{ and } n = |F|$$

We now define how fault environments are composed together with fault-prone systems. Some preliminary considerations are needed. We need to model the physical modification performed by the environment on the faulty part of the system. For this, a function flip is defined as $\mathsf{flip}(S, L) = S[l \mapsto \neg S[l], l \in L]$, which gives the result of flipping the value of every location in L of state S (assuming $L \subseteq F$). Notice that when L is the empty set, no modification to the state is performed. We also need to formalize that an attacker can distinguish only a subset of all possible actions of the system. This is obtained by assuming that there is a function $low \in Act \rightarrow LAct \cup \{\tau\}$ that behaves as the identity for actions in $LAct$, and maps any other action to τ. This provides the public view of the system's output. Finally, we say that a state S is *stuck* if there is no transition from that state.

Definition 3 (Fault-prone System and Environment): Consider a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$, and an environment $Env = (Err, \mathsf{Fault})$ where $Err = \langle \mathcal{E}, LAct \cup \{\tau\}, \rightsquigarrow \rangle$. The composition of Sys and Env, defined as $Sys \times Env = \langle \mathcal{S} \times \mathcal{E}, (Act, [0,1]), \rightarrow \rangle$ where $\mathcal{S}$ is the set of all possible states for Sys, defines a labelled transition system whose states are pairs of the system state and the environment state, and with transitions labelled with an action a and a probability p, written $\overset{a}{\rightarrow}_p$. Transitions depend on the state of the system as follows:

- If the system state S is not stuck, it undergoes a transient fault according to the flip function; then, providing the flipped state is not stuck, the execution takes place, namely

$$\textbf{Step}\ \frac{\pi = \{L \mid L \in \wp(F) \text{ and } \mathsf{flip}(S, L) \overset{a}{\rightarrow} S'\} \qquad p = \Sigma_{L \in \pi} \mathsf{Fault}(E)(L) \qquad E \overset{low(a)}{\rightsquigarrow} E'}{\langle\ S,\ E\ \rangle \overset{a}{\rightarrow}_p \langle\ S',\ E'\ \rangle}$$

  Observe that, in general, there might be several subsets of $L \subseteq F$ such that $\mathsf{flip}(S, L)$ results in a state that (i) performs the same action a and (ii) performs a transition to the final state S'. For this reason, the probability associated with the rule corresponds to the sum of all probabilities associated with locations in the set π.

- If the system state is made stuck by a transient fault, or it is already stuck, it does not perform any action. However, both the environment and the composition of the system state with the environment state evolve as if the system had performed a τ action.

$$\textbf{Stuck-1}\ \frac{\exists l S \overset{l}{\rightarrow} S' \qquad L \in \wp(F) \qquad \mathsf{flip}(L, S) \not\rightarrow \qquad p = \mathsf{Fault}(E)(L) \qquad E \overset{\tau}{\rightsquigarrow} E'}{\langle\ S,\ E\ \rangle \overset{\tau}{\rightarrow}_p \langle\ \mathsf{flip}(L, S),\ E'\ \rangle}$$

$$\textbf{Stuck-2}\ \frac{S \not\rightarrow \qquad E \overset{\tau}{\rightsquigarrow} E'}{\langle\ S,\ E\ \rangle \overset{\tau}{\rightarrow}_1 \langle\ S,\ E'\ \rangle}$$

  We enforce these restrictions because an error environment should not be able to distinguish between an active but "silent" and a stuck state. Notice that this way of modeling the composition of systems and environments guarantees that any state of the composition can progress.

This form of transition system is sometimes known as a *fully probabilistic labelled transition system*, or a *labelled markov process*.

B. Defining Security

We can now reason about security of a system operating in an environment: $Sys \times Env$. Firstly, we define the observations that the attacker can perform. Then, we define when sensitive data remains secure despite the attacker observations.

Our attacker sees sequences of actions in $LAct \cup \{\tau\}$, called *traces*, and measures their probability, but does not otherwise have access to the state of the fault-prone system.

We say that the sequence $r = Z \overset{a_0}{\rightarrow}_{p_0} Z_1 \ldots Z_{n-1} \overset{a_{n-1}}{\rightarrow}_{p_{n-1}} Z_n$ is a *run* of size n of a system state $Z \in Sys \times Env$ and has probability $\mathsf{Pr}(r) = \Pi_{0 \le i \le n-1} p_i$. The set of all n-runs of Z are denoted $\mathsf{run}_n(Z)$, and we define the set of all runs for Z as $\mathsf{run}(Z) = \cup_n \mathsf{run}_n(Z)$. Consider a run $r = Z \overset{a_0}{\rightarrow}_{p_0} Z_1 \ldots Z_{n-1} \overset{a_{n-1}}{\rightarrow}_{p_{n-1}} Z_n$ and let $\mathsf{trace} \in \mathsf{run}(Z) \rightarrow (LAct \cup \{\tau\})^*$ be a function such that $\mathsf{trace}(r) = low(a_0) \ldots low(a_{n-1})$. For any $t \in (LAct \cup \{\tau\})^n$, define

$\mathsf{Pr}_Z(t) = \Sigma_{\{r \in \mathsf{run}_n(Z) | \mathsf{trace}(r)=t\}} \mathsf{Pr}(r)$. This definition induces a probability distribution over $(LAct \cup \{\tau\})^n$.

Proposition 1: For any state $Z \in Sys \times Env$ and for any $n \geq 0$ $\Sigma_{t \in (LAct \cup \{\tau\})^n} \mathsf{Pr}_Z(t) = 1$.

Proof 1: Appendix A-A.

We can now define the notion of *probabilistic noninterference* [17] that characterises security for fault-prone systems. For this purpose, we consider the case when the initial state of a fault-prone system is an encoding of three different components: (i) a "program", the set of instructions executed by the system, (ii) "public data", that stores values known by an attacker and (iii) "private data" for confidential information. We formalize this partition by defining three mutually disjoint sets $ProgLoc$, $LowLoc$ and $HighLoc$ (such that $Loc = ProgLoc \cup LowLoc \cup HighLoc$) and, for a system state S, by defining the program component P as $S\big|_{ProgLoc}$, the public data L as $S\big|_{LowLoc}$ and H as $S\big|_{HighLoc}$.

Observe that the way locations are partitioned between program and data is orthogonal to the way they are partitioned between fault-tolerant and faulty components. This is because fault-tolerance is orthogonal to the way security is defined.

Our definition of security, *Probabilistic Fault-Resilient Non-Interference* (PNI), requires a notion of equivalence to be defined for states that look the same from an attacker's point of view. Two system states S and S' are low equivalent, written as $S =_L S'$, if $S\big|_{LowLoc} = S'\big|_{LowLoc}$. Low equivalence provides us a way for defining when the program component of a state is secure even in the presence of transient faults. Intuitively the definition says that a program component P is secure if the observed probability for any trace is independent of the sensitive data, for all system states where P is the program component.

Definition 4 (PNI): Let Sys be a fault-prone system and let P be a program component of Sys. We say that P is *probabilistic fault-resilient non-interfering*, if for any system states $S, S' \in Sys$ such that $S'\big|_{ProgLoc} = S\big|_{ProgLoc} = P$ and $S =_L S'$, it holds that for any state E of any environment Env, for any $n \geq 0$ and for any $t \in (LAct \cup \{\tau\})^n$ we have $\mathsf{Pr}_{\langle\ S,\ E\ \rangle}(t) = \mathsf{Pr}_{\langle\ S',\ E\ \rangle}(t)$.

The definition demands that probability of publicly observable traces only depends on values stored in the low locations. Also, it requires that this must hold for any fault-environment.

C. Possibilistic Characterisation of Fault-Resilient Non-Interference

Reasoning directly about fault-resilient non-interference is difficult because (i) it demands reasoning about probabilities, and (ii) it quantifies over all possible attackers (in a given class). In this section, we address the first of these problems.

A *possibilistic* model (i.e. not probabilistic) of the interaction between a fault-prone system and the error environment can be obtained by interleaving the transitions of the fault prone system with a nondeterministic flipping of zero or more bits. While this model avoids reasoning about probability distributions as well as injection of faults by an attacker, it is not adequate to directly capture security, as it is well-known that possibilistic noninterference suffers from probabilistic information leaks (see e.g. [17]). In order to capture PNI precisely, we augment this transition system by making the location of the faults observable.

Definition 5 (Augmented Fault-prone System): Given a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$ we define the augmented system Sys^+ as $Sys^+ = \{Loc, \wp(F) \times Act, \rightsquigarrow\}$, where $\rightsquigarrow$ is defined according to two cases:

- If the system state S is not stuck, it undergoes a nondeterministic transient fault first; then, providing the flipped state is not stuck, execution takes place, namely

$$\frac{\mathsf{flip}(S, L) \xrightarrow{a} S' \quad L \subseteq F}{S \overset{L,a}{\rightsquigarrow} S'}$$

  Observe that, compared to the corresponding rule in Definition 3, we have that L induces a unique transition since it appears in the transition label.

- If the system state is stuck, or it is made stuck by a transient fault, the transition does not modify it. However, the label attached to the transition is (L, τ) so that, as in Definition 3, we make a stuck state indistinguishable from a silently diverging one.

The model resembles the composition of fault-prone systems and fault environments presented in Definition 3, including the fact that it hides the termination of a system configuration. However, it introduces two main differences that influence the way security is defined. On one hand, the augmented system is purely nondeterministic, and this supports a simpler definition of security. On the other hand, the augmented system has more expressive labels, that include not only the action performed by the system but also information about flipped locations.

We call the sequence $r = S \overset{L_0,a_0}{\rightsquigarrow} S_1 \ldots S_{n-1} \overset{L_{n-1},a_{n-1}}{\rightsquigarrow} S_n$ a *possibilistic run* of a system state $S \in Sys^+$, and we say that $t = L_0, low(a_0), \ldots, L_{n-1}, low(a_{n-1})$ is its corresponding trace. We write $S \overset{t}{\rightsquigarrow\!\!\!\!\rightarrow}$ when there exists a run r, produced by S, that corresponds to t. With these conventions, security for augmented systems, *Possibilistic Fault-Resilient Non-Interference* (PoNI), is defined by using the same notation presented in the previous section for fault-prone systems composed with environments.

Definition 6 (PoNI): We say that a program component P satisfies *possibilistic fault-resilient non-interference*, if for any $S, S' \in Sys^+$ such that $S'\big|_{ProgLoc} = S\big|_{ProgLoc} = P$ and $S =_L S'$, for any trace t, it holds that $S \overset{t}{\rightsquigarrow\!\!\!\!\rightarrow} \Leftrightarrow S' \overset{t}{\rightsquigarrow\!\!\!\!\rightarrow}$.

The following result says that the definitions of PNI and PoNI coincide.

Theorem 1: A program component P satisfies PNI if and only if it satisfies PoNI.

Proof 2: Appendix A-C.

Some intuition about this result is perhaps appropriate here. As mentioned previously, it is common wisdom that the possibilistic view of a system's behaviour may not be adequate

to rule out information flows transmitted not by the *possible* observable behaviours, but by the probability of their occurrence [17]. At first glance this seems contrary to the thrust of Theorem 1, where a probabilistic security property is implied by a possibilistic property and not vice versa. The reason why this works is that we augmented the original model with additional observable behaviours (the exact fault locations); the information carried by this additional information subsumes (and hence is a proxy for) the information that can be carried solely by probabilities.

D. Strong Security Implies PNI

We now formalize a different notion of security that guarantees PoNI (and hence PNI) without explicitly modeling the effects of transient faults in a fault-prone system. This notion, called *Strong Security*, was developed as a way to capture a notion of scheduler independent compositional security for multithreaded programs [30].

Strong security is a bisimulation relation over program components of fault-prone systems. Our goal is to relate strong security to the possibilistic security definition established for augmented systems, and show that indeed it is stronger. Before doing so, we need to make sure that the semantics of fault-prone systems hides termination, as it is the case for augmented systems. In particular, for a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$, we define its termination-transparent version as $Sys^{\infty} = \{Loc, Act, \rightarrow_{\infty}\}$ where $\rightarrow_{\infty}$ coincides with $\rightarrow$ for active states, but has additional transitions $S \xrightarrow{\tau}_{\infty} S$ whenever $S \not\rightarrow$ (details are discussed in Appendix A-B). With this, we define strong security for termination-transparent fault-prone systems as follows.

Definition 7 (Strong Security (SS)): Let Sys be a fault-prone system and $Sys^{\infty} = \{Loc, Act, \rightarrow_{\infty}\}$ be the corresponding termination-transparent fault-prone system. A symmetric relation R between program components is a strong bisimulation if for any $(P, P') \in R$ we have that for any two states S, V in $\mathcal{S}$, if $S\big|_{ProgLoc} = P$ and $V\big|_{ProgLoc} = P'$ and $S =_L V$ and $S \xrightarrow{a}_{\infty} S'$ then $V \xrightarrow{b}_{\infty} V'$ such that (i) $low(a) = low(b)$ and (ii) $S' =_L V'$ and (iii) $(S'\big|_{ProgLoc}, V'\big|_{ProgLoc}) \in R$. We say that a program component P is strongly secure if there exists a strong bisimulation R such that $(P, P) \in R$.

Intuitively, a program component P is strongly secure when differences in the private part of the data are neither visible in the computed public data, nor in the transition label. This anticipates the fact that even though an external agent (the error environment, in our scenario) might alter the data component, the program behavior does not reveal anything about secrets.

The next result shows that Strong Security is sufficient to obtain PoNI. Notice, however, that the definition of Strong Security only deals with the modifications that occur in the data part of a fault-prone system. Hence, it only makes sense for the class of systems that host the program component in the fault-tolerant part of the configuration.

Theorem 2 (Strong security $\Rightarrow$ PoNI): Let P be a program component such that $ProgLoc \subseteq T$. If P satisfies SS then P satisfies PoNI.

Proof 3: Appendix A-D.

We now propose a brief overview of the strategy that is used to prove this result.

The main step when proving this theorem is to find a formalization of SS that is closer to the way PoNI is defined, and use it to show that SS is indeed stronger than PoNI.

We achieve this result by introducing a "transition traces" semantics for fault-prone systems. This semantic model has been proposed in [9] for assigning a trace-based semantics to concurrent programs. We say that a program P_0 has a transition trace $(M_0, a_0, N_0), (M_1, a_1, N_1), \ldots, (M_k, a_k, N_k), \ldots$ if, for any i, the program P_i transforms the input data configuration M_i into the output data configuration N_i, with a visible action a_i, and becomes P_{i+1}. Notice that the output data configuration resulting from a computation step does not necessarily correspond to the input data configuration used in the subsequent step. This feature explicitly models the fact that in a concurrent setting, the actions of a program can be arbitrarily interleaved with the modification performed by the environment in which the execution takes place.

We instantiate the transition trace definition in the context of fault-prone system by considering the partition of the system locations between the program and the data component (in its public and private parts) introduced for defining Strong Security. Notice that, in fact, this semantic view of fault-prone systems is reminiscent of the way Strong Security is defined, since in both cases the memory configurations are changed in every step.

Within the transition trace semantic model for fault-prone systems, we define a security property called Strong Trace-based Security, that captures the main features of Strong Security and, at the same time, models the modification induced by bit flips. We say that a program component of a fault-prone system satisfies Strong Trace-based Security when any two transition traces that are input-low equivalent (i.e. all the corresponding input configurations are low equivalent), are also output low equivalent (i.e. all the corresponding output configurations, and all the corresponding actions, are low equivalent).

We use transition trace semantics and Strong Trace-based Security to prove the relation between SS and PoNI. The first step is to show that Strong Security implies Strong Trace-based Security. This is intuitively correct since the latter property is, approximately, an unwinding of the former. Then, we show that Strong Trace-based Security implies PoNI. This is achieved by noticing that low equality is preserved by bit-flips, hence any pair of runs involved in PoNI can be mapped to their correspondent pair involved in the definition of Strong Trace-based Security.

We conclude this section by noting a negative result: PoNI does not imply SS. Strong security requires that the partition of the state into program, low, and high part is preserved (respected) during the whole computation, whereas the definition of PoNI imposes no special requirement on intermediate states

of the computation.

III. A Type System for Fault-Resilient Non-Interference

In this section, we present an enforcement mechanism capable of synthesizing strongly secure assembly code. The enforcement is given as a type-directed compilation from a source while-language. All in all, we present a concrete instance of our fault-prone system formalization by defining the RISC architecture (Section III-A) and propose a technique to enforce strong security over RISC programs (Section III-B), whose soundness is sketched in Section III-C. By achieving strong security, we automatically get secure RISC code that is robust against transient faults (recall Theorem 2).

A. A Fault-Prone RISC Architecture

The architecture we are interested in has various hardware components to operate over data and instructions.

Data resides in the memory and in the register bank. We model the memory $\mathcal{M}$ as a function $\mathbb{W} \to \mathbb{W}$, where $\mathbb{W}$ is the set of all constants that can be represented with a machine word. The register bank R is modeled as a function $Reg \to \mathbb{W}$, where Reg, ranged over by r, r', is a set of register names.

$$
\begin{array}{lcl}
I & ::= & [l:]B \\
B & ::= & \mathsf{load}\ r\ k \mid \mathsf{store}\ k\ r \mid \mathsf{jmp}\ l \mid \mathsf{jz}\ l\ r \mid \mathsf{nop} \\
 & & \mathsf{movek}\ r\ n \mid \mathsf{mover}\ r\ r' \mid \mathsf{op}\ r\ r' \mid \mathsf{out}\ ch\ r \\
ch & ::= & low \mid high
\end{array}
$$

Fig. 1. RISC instructions syntax

Figure 1 describes the instruction set of our architecture. We consider that every instruction I could be optionally labeled by a label in the set Lab. Instruction $\mathsf{load}\ r\ k$ accesses the data memory $\mathcal{M}$ with the pointer $k \in \mathbb{W}$ and writes the value pointed by k into register $r \in Reg$. The corresponding $\mathsf{store}\ k\ r$ instruction writes the content of r into the data memory address k. Instruction $\mathsf{jmp}\ l$ causes the control-flow to transfer to the instruction labeled as l. Instruction $\mathsf{jz}\ l\ r$ performs the jump only if the content of register r is zero. Instruction nop performs no computation. The instruction $\mathsf{movek}\ r\ k$ writes the constant k to r, whereas the instruction $\mathsf{mover}\ r\ r'$ copies the content in r' to r. The instruction op stands for a generic binary operator that combines values in r and r' and stores the result in r. Instruction $\mathsf{out}\ ch\ r$ outputs the constant contained in r into the channel ch, that can be either low or $high$.

The processor fetches RISC instructions from the code memory P, separated from $\mathcal{M}$. The code memory is modeled as a list of instructions. We require the code memory to be well-formed, namely not having two different instructions labeled in the same way. A dedicated *program counter* register stores the location in P hosting the instruction being currently executed. The value of the program counter is ranged over by pc.

As described in Section II, we partition the architecture into faulty and fault-tolerant components. Both R and $\mathcal{M}$ are considered to be faulty: transient faults can strike any location at any time during the execution. On the other hand, we assume P and the program counter are implemented in the fault-tolerant part of the architecture. The fact that the code memory is fault-tolerant corresponds to having the machine code in a read-only memory with ECC, a common assumption in dependability domain. The requirement on the program counter is a restriction that turns out to be necessary for proving the soundness of our enforcement mechanism.

$$
\textbf{Load}\ \frac{P(pc) = \mathsf{load}\ r\ p}{\langle P, pc, R, \mathcal{M}\rangle \xrightarrow{\tau} \langle P, pc^+, R[r \mapsto \mathcal{M}(p)], \mathcal{M}\rangle}
$$

$$
\textbf{Store}\ \frac{P(pc) = \mathsf{store}\ p\ r}{\langle P, pc, R, \mathcal{M}\rangle \xrightarrow{\tau} \langle P, pc^+, R, \mathcal{M}[p \mapsto R(r)]\rangle}
$$

$$
\textbf{Jz-S}\ \frac{P(pc) = \mathsf{jz}\ l\ r \quad R(r) = 0}{\langle P, pc, R, \mathcal{M}\rangle \xrightarrow{\tau} \langle P, pc \mapsto \mathsf{res}_P(l), R, \mathcal{M}\rangle}
$$

$$
\textbf{Jz-F}\ \frac{P(pc) = \mathsf{jz}\ l\ r \quad R(r) \neq 0}{\langle P, pc, R, \mathcal{M}\rangle \xrightarrow{\tau} \langle P, pc^+, R, \mathcal{M}\rangle}
$$

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

Fig. 2. Selected rules for RISC instructions semantics

We instantiate the RISC architecture as a fault-prone system by defining the semantics of the language as a labelled transition system. A state is defined as a quadruple $\langle P, pc, R, \mathcal{M}\rangle$, for $pc \in \mathbb{W}$, where the first two elements correspond to the fault-tolerant portion of the hardware. Any action of the system is either an output ($low!k$ when the output is performed on the publicly observable channel, $high!k$ otherwise) or the silent action τ. A few examples of transition rules are described in Figure 2 (the full presentation can be found in the extended version of this paper [13]). We write $P(pc)$ as a shorthand for the instruction at position pc in P and pc^+ as a shorthand for $pc + 1$. We assume that the function $\mathsf{res}_P \in Lab \rightharpoonup \mathbb{W}$ returns the position at which label l occurs in P: $\mathsf{res}_P(l) = i$ iff $P(i) = l : B$ for some B.

Once the rule [**Load**] is triggered, the register content at r is updated with the memory content at p ($R[r \mapsto \mathcal{M}(p)]$), and the program counter is incremented by one (pc^+). Conversely, the rule [**Store**] writes the content at register r in memory location p ($\mathcal{M}[p \mapsto R(r)]$). In case of a jump instruction, neither memory nor registers are modified. If the guard is 0, as in rule [**Jz**$-$**S**], the execution is restarted at the instruction of the label used as the jump argument $pc \mapsto \mathsf{res}_P(l)$, otherwise the program counter is just incremented. All previous instructions map to silent actions τ: channels are written by instruction [**Out**] which, on the other hand, leaves both register and memory untouched.

B. Strong Security Enforcement

We guarantee strong security for some RISC programs via a novel approach based on type-directed compilation.

Our strategy targets a simple high-level imperative language, `while`. For `while` programs we define a type system that performs two tasks: (i) translation of `while` programs into `RISC` programs (ii) enforcement of Strong Security on `while` programs. The compilation is constructed so that the strong security at the level of `while` programs is preserved by the compilation.

To facilitate the proof of strong security, the method is factored into a two-step process: a type-directed translation to an intermediate language followed by a simple compilation to `RISC`. In this section, we limit ourself to an overview of the method and therefore elide this intermediate step from the presentation.

The grammar of the `while` language is presented in Figure 3. Both expressions and commands are standard, and assume that the language contains an output command out.

$$\begin{array}{rcllll} C & ::= & \mathsf{skip} & \mid & x := E & \mid & \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid & \mathsf{out}\ ch\ E & \mid & C_1; C_2 & \mid & \mathsf{while}\ x\ \mathsf{do}\ C \\ E & ::= & k \in \mathbb{W} & \mid & x \in \mathit{Var} & \mid & E_1\ \mathsf{op}\ E_2 \\ ch & ::= & \mathit{low} & \mid & \mathit{high} & & \end{array}$$

Fig. 3. `while` programs syntax

Typing Expressions The general structure of the typing judgement for expressions is presented in Figure 4.

$$\underbrace{\Phi, A, l}_{(1)} \Vdash \overbrace{E}^{(2)} \hookrightarrow \underbrace{P}_{(3)}, \overbrace{\langle \lambda, n \rangle}^{(4)}, \underbrace{r, \Phi'}_{(5)}$$

Fig. 4. General structure for expression typing rules

The core part of the judgement says that `while` expression E (2) can be compiled to secure `RISC` program P (3), with security annotation (4). For defining the security annotation, we assume `while` variables and `RISC` registers are partitioned into two security levels $\{L, H\}$ (ordered according to $\sqsubseteq$, which is the smallest reflexive relation for which $L \sqsubseteq H$) according to the function $\mathsf{level} \in \mathit{Var} \cup \mathit{Reg} \to \{L, H\}$. The first component λ of a security annotation (λ, n) specifies the security level of the registers that are used to evaluate an expression. The second component n represents the number of `RISC` computation steps that are necessary to evaluate E. The reason for tracking this specific information about expressions (and in commands later on) is related to obtain assembly code which avoids timing leaks – the type-directed compilation will use n, for instance, to do *padding* of code when needed [1]. In fact, most of the involved aspects in our type-system arises from avoiding timing leaks in low-level code.

Because the compilation is defined compositionally, some auxiliary information (1) is required: we firstly need to know the label l which is to be attached to the first instruction of P. Also, we have to consider the set A of registers which cannot be used in the compilation of E, since they hold intermediate values that will be needed after the computation of E is complete. Finally, we need to keep track of how variables are mapped to registers. This is done via the register record Φ, which requires a slightly more elaborate explanation.

Register Records In order to allow for efficiently compiled code, expression compilation builds up a record of associations between `while` variables and `RISC` registers in a *register record* $\Phi \in \mathit{Reg} \rightharpoonup \mathit{Var}$. If $\Phi(r) = x$ then it means that the current value of variable x is present in register r. There are two important aspects of the register record to consider. Firstly, the domain of Φ is finite and roughly corresponds to the largest number of variables used in any expression. Notice that by pre-processing the code before type checking this can be reduced to a fixed size - and so "register shuffling" can thus be represented in the source-code prior to compilation. Secondly, observe that the register record produced by a compilation is highly nondeterministic, meaning that we do not build any particular register allocation mechanism into the translation. In this context nondeterminism is used to keep the specification simple by not committing to any particular choice, and thus all available choices are shown to be secure.

The rules (in particular the later rules for commands) involve a number of operations on register records which we briefly describe here. We ensure that a register record is always a partial bijection, namely a register is associated to at most one variable and a variable is associated with at most one register. We write $\Phi[r \leftrightarrow x]$ to denote the minimal modification of Φ which results in a partial bijection mapping r to x. Similarly, $\Phi[r \not\rightarrow]$ denotes the removal of any association to r in Φ. The intersection of records $\Phi \sqcap \Phi'$ is just the subset of the bijections on which Φ and Φ' agree. Finally, inclusion between register records, written as $\Phi \sqsubseteq \Phi'$, holds if all associations in Φ are also found in Φ'.

Beside the compiled expression and its security annotation, each rule returns a modified register record and specifies the actual register where the evaluation of the expression E is found at the end of the execution of P (5).

Expression Rules

It will be convenient to extend the set of instructions with the empty instruction ϵ_I. Some sample rules for computing the types of expressions are presented in Figure 5 (the full presentation can be found in the extended version of this paper [13]).

$$\textbf{K}\ \frac{r \notin A}{\Phi, A, l \Vdash k \hookrightarrow [l :\ \mathsf{movek}\ r\ k], \langle \mathsf{level}(r), 1 \rangle, r, \Phi[r \not\rightarrow]}$$

$$\textbf{V-cached}\ \frac{\Phi(r) = x}{\Phi, A, l \Vdash x \hookrightarrow [l :\ \epsilon_I], \langle \mathsf{level}(r), 0 \rangle, r, \Phi}$$

Fig. 5. Selected type system rules for `while` expressions

In rule $[\mathbf{K}]$ the constant k is compiled to code which writes the constant to some register r via the $\mathsf{movek}\ r\ k$ instruction, providing r is not already in use ($r \notin A$). As a result, any previous association between register r and a variable is lost ($\Phi[r \not\rightarrow]$). The security level of the result is simply the level of the register, and the computation time is one.

$$\underbrace{\Phi, l}_{(1)} \vdash C \hookrightarrow P, \overbrace{\langle w, t\rangle}^{(3)}, \underbrace{l', \Phi'}_{(2)}$$

Fig. 6. General structure for command typing rules

In rule $[\mathbf{V-cached}]$ the variable to be compiled is already associated to a register, hence no code is produced.

Typing Commands The general structure of a typing rule for commands is presented in Figure 6. Judgements for commands assume a starting label for the code to be produced, and an incoming register record (1). A compilation will result in a new (outgoing) register record, and the label of the next instruction following this block (2) (cf. Figures 8, 9 and 10). The security annotation (3) is similar to that for expressions; w, the *write effect*, provides information about the security level of variables, registers, and channels to which the compiled code writes, and t describes its timing behaviour. However, w and t are drawn from domains which include possible uncertainty.

The write effect w is described with a label taken from the two-element set $\{\mathsf{Wr}\ H, \mathsf{Wr}\ L\}$, with partial ordering $\mathsf{Wr}\ H \sqsubseteq \mathsf{Wr}\ L$. The value $\mathsf{Wr}\ H$ is for programs that never write to registers and memory locations outside of H. The value $\mathsf{Wr}\ L$ is used when write operations might occur at any security level.

$$\mathsf{Trm}\ 0 \sqsubseteq \cdots \sqsubseteq \mathsf{Trm}\ n \sqsubseteq \cdots \sqsubseteq \mathsf{Trm}\ L \sqsubseteq \mathsf{Trm}\ H,\ n \in \mathbb{N}$$

$$t_1 \sqcup t_2 = \begin{cases} \mathsf{Trm}\ L & \text{if } t_1 \sqsubseteq \mathsf{Trm}\ L \text{ and } t_2 \sqsubseteq \mathsf{Trm}\ L \\ \mathsf{Trm}\ H & \text{otherwise} \end{cases}$$

$$t_1 \uplus t_2 = \begin{cases} \mathsf{Trm}\ n_1 + n_2 & \text{if } \forall i \in \{1,2\} \quad t_i = \mathsf{Trm}\ n_i \\ t_1 \sqcup t_2 & \text{otherwise} \end{cases}$$

Fig. 7. Termination Partial Ordering

The timing behavior of a command is described by an element of the partial order (and associated operations) defined in Figure 7. We use timing $t = \mathsf{Trm}\ n$, for $n \in \mathbb{N}$, when termination of the code is guaranteed in exactly n steps (and hence is independent of any secrets); $t = \mathsf{Trm}\ L$ is used for programs whose timing characterization does not depend on secret values, but whose exact timing is either unimportant, or difficult to calculate statically. When secrets might directly influence the timing behavior of a program, the label $\mathsf{Trm}\ H$ is used.

Command Rules We now introduce some of the actual rules for commands. The concatenation of code memories P and P' is written $P +\!\!+ P'$ and is well defined if the resulting program remains well-formed. It will be convenient to extend the set of labels Lab with a special empty label ϵ_{lab} such that $\epsilon_{lab} : B$ simply denotes B. Also, we consider that the empty instruction ϵ_I is such that if $P = [B, I_1, \ldots, I_n]$ we define $[l : \epsilon_I] +\!\!+ P$ as $[l : B, I_1, I_2, \ldots, I_n]$.

The rule $[:=]$ in Figure 8 requires that the security

$$:= \frac{\Phi, \{\}, l \Vdash E \hookrightarrow P, \langle \mathsf{level}(x), n\rangle, r, \Phi' \qquad \Phi'' = \Phi'[r \leftrightarrow x]}{\Phi, l \vdash x := E \hookrightarrow \{P\ +\!\!+\ [\mathsf{store}\ v2p(x)\ r]\},\ tp,\ \epsilon_{lab},\ \Phi''}$$

$$\text{where } tp = \begin{cases} \langle \mathsf{Wr}\ H, \mathsf{Trm}\ n+1\rangle & \text{if } \mathsf{level}(x) = H \\ \langle \mathsf{Wr}\ L, \mathsf{Trm}\ L\rangle & \text{otherwise.} \end{cases}$$

Fig. 8. Type system rule for assignment

level of expression E matches the level of the variable x ($\langle \mathsf{level}(x), n\rangle$). If this is possible, the compilation is completed by storing the value of r into the pointer corresponding to x via the instruction $\mathsf{store}\ v2p(x)\ r$ (assuming there exists an injective function $v2p \in Var \to \mathbb{W}$ which maps `while` variables to memory locations), and the register record is updated by associating r and x ($\Phi'[r \leftrightarrow x]$). The resulting security annotation depends on the level of x: when $\mathsf{level}(x) = H$ the security annotation is $\langle \mathsf{Wr}\ H, \mathsf{Trm}\ n+1\rangle$, otherwise it is $\langle \mathsf{Wr}\ L, \mathsf{Trm}\ L\rangle$. Rules for skip and out can be found in the extended version of this paper [13].

The rule $[\mathsf{if} - \mathbf{any}]$ in Figure 9 builds the translation of the if statement by joining together several `RISC` fragments. The basic idea of this rule is that it follows Denning's classic condition for certifying information flow security [14]: if the conditional involves high data then the branches of the conditional cannot write to anything except high variables. This is obtained by imposing the side condition $w_i \sqsubseteq \mathsf{write}(\lambda)$, where w_i is the write-effect of the respective branches, and write is a function mapping the security level of the guard into its corresponding write-effect (such that $\mathsf{write}(H) = \mathsf{Wr}\ H$ and $\mathsf{write}(L) = \mathsf{Wr}\ L$). In this rule, in contrast to the $[\mathsf{if} - \mathbf{H}]$ rule for the conditional, the timing properties of the two branches may be different, so we do not attempt to return an accurate timing. Hence, the use of the operator $\sqcup$ which just records whether the timing depends on only low data, or possibly any data (notice that the security level of the guard is mapped into its corresponding timing label by the function term, such that $\mathsf{term}(L) = \mathsf{Trm}\ L$ and $\mathsf{term}(H) = \mathsf{Trm}\ H$). The compilation of the conditional code into `RISC` is fairly straightforward: compute the expression (P_0) into register r, jump to else-branch (P_2) if r is zero, otherwise fall through to "then" branch (P_1) and then jump out of the block. The resulting register record of the whole command compilation is the common part of the register records resulting from the respective branches.

The rule $[\mathsf{if} - \mathbf{H}]$ (Figure 9) allows the system to be more permissive. It deals with a conditional expression which only writes to high locations – a so-called *high conditional*. This rule, when applicable, compiles the high conditional in a way that guarantees that its timing behaviour is *independent of the high data*. This is important since it is the only way that we can permit a computation to securely write to low variables after a high conditional. This is related to timing-sensitive information-flow typing rules for high conditionals by Smith [32]. The basic strategy is to compute the timing

$$\textbf{if-any}\ \frac{\Phi, \{\}, l \Vdash E \hookrightarrow P_0, \langle \lambda, n_0\rangle, r, \Phi_1 \quad \forall i \in \{1,2\} \quad \Phi_1, \epsilon_{lab} \vdash C_i \hookrightarrow P_i,\ \langle w_i, t_i\rangle,\ l_i,\ \Phi_{i+1} \quad w_i \sqsubseteq \mathsf{write}(\lambda) \quad br, ex \text{ fresh}}{\Phi, l \vdash \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \left\{ \begin{array}{l} P_0 +\!\!+ [\mathsf{jz}\ br\ r] +\!\!+ \\ P_1 +\!\!+ [l_1 :\ \mathsf{jmp}\ ex] +\!\!+ \\ br : P_2 +\!\!+ [l_2 :\ \mathsf{nop}] \end{array} \right\},\ \langle \mathsf{write}(\lambda), \mathsf{term}(\lambda) \sqcup t_1 \sqcup t_2\rangle,\ ex,\ \Phi_2 \sqcap \Phi_3}$$

$$\textbf{if-H}\ \frac{\begin{array}{c}\Phi, \{\}, l \Vdash E \hookrightarrow P_0, \langle H, n_0\rangle, r, \Phi_1 \quad \forall i \in \{1,2\} \quad \Phi_1, \epsilon_{lab} \vdash C_i \hookrightarrow P_i,\ \langle \mathsf{Wr}\ H, \mathsf{Trm}\ n_i\rangle,\ l_i,\ \Phi_{i+1} \\ m = n_0 + max(n_1, n_2) + 2 \qquad br, ex \text{ fresh}\end{array}}{\Phi, l \vdash \mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \left\{ \begin{array}{r} P_0 +\!\!+ [\mathsf{jz}\ br\ r] +\!\!+ P_1 +\!\!+ l_1 : \mathsf{nop}^{n_2 - n_1} +\!\!+ [\mathsf{jmp}\ ex] \\ +\!\!+ P_2 +\!\!+ l_2 : \mathsf{nop}^{n_1 - n_2} +\!\!+ [\mathsf{nop}] \end{array} \right\},\ \langle \mathsf{Wr}\ H, \mathsf{Trm}\ m\rangle,\ ex,\ \Phi_2 \sqcap \Phi_3}$$

Fig. 9. Type rules for if

of each branch (n_1 and n_2 respectively) and pad the respective branches in the compiled code with sequences of nop instructions so that they become equally long, where nop^m is a sequence of m consecutive nop instructions when $m > 0$, and is ϵ_I otherwise.

The rule [;] for sequential composition (Figure 10) is largely standard: the label and register records are passed sequentially from inputs to outputs, and the security types are combined in the obvious way. The only twist, the side condition, encodes the key idea in the type system of Smith [32]. If the computation of the first command has timing behaviour which might depend on high data ($t_1 = \mathsf{Trm}\ H$), then the second command cannot be allowed to write to low data ($w_2 = \mathsf{Wr}\ H$), as this would otherwise reveal information about the high data through timing of low events.

The compilation of the while command (Figure 10) is quite involved for two reasons. Firstly, as one would expect in a typing rule for a looping construct, there are technical conditions relating the register record at the beginning of the loop, and the register record on exit. This is because we need a single description of the exit register record Φ_B which approximates both the register record at the start of the loop body ($\Phi[r \leftrightarrow x]$) and the register record after computing the loop body and putting x back into register r ($\Phi_E[r \leftrightarrow x]$). Secondly, for technical reasons relating purely to the proof of correctness (security), the code is (i) a little less compact that one would expect to write due to an unnecessarily repeated subexpression, and (ii) contains a redundant instruction store $v2p(x)$ r immediately after having loaded x into r. The lack of compactness is due to the fact that the proof goes via an intermediate language that cannot represent the ideal version of the code. The redundant instruction establishes a particular invariant that is needed in the proof: not only is x in register r, but it arrived there as the result of writing r into x. The security concerns are taken care of by ensuring that the security level of the whole loop is consistent with the levels of the branch variable x and the branch register r, and that if the timing of the body might depend on high data, then the level of the loop variable (and hence the whole expression) must be H.

C. Soundness

In this section we give a brief outline of the correctness proof of the type systems, the full details are provided in the extended version of this paper.

We begin by instantiating the definition of strong security for RISC programs, which requires to view the RISC machine as an instance of a fault-prone system (Definition 1). For this we consider the set of locations *Loc* of the RISC fault-prone system to be the names of the individual bits comprising the registers and memories. So, for example, a general purpose register r corresponds to some set of locations $r_0, \ldots r_{31}$ (for a word-size of 32). With this correspondence, the set of states of the RISC system are isomophic to the set of functions $Loc \to \{0, 1\}$. As mentioned earlier, the fault-prone locations F are those which correspond to the general purpose registers and the data memory.

For the definition of security we must additionally partition the locations into the program *ProgLoc*, the low locations *LowLoc*, and the high locations *HighLoc*: *ProgLoc* comprises the locations of the code *and* the program counter register, *LowLoc* the locations of the low variables and registers, and *HighLoc* the locations of the high variables and registers.

Since assembly programs are run starting at their first instruction, the following slightly specialised version of strong security is appropriate:

Definition 8 (Strong Security for RISC programs): We say that an assembly program P is strongly secure if $(P, 0)$ is strongly secure according to Definition 7 instantiated on the fault-prone system $\mathcal{A} = \{Loc, \{ch!k | ch \in \{low, high\}$ and $k \in \mathbb{W}\} \cup \{\tau\}, \to\}$.

The type system defined in Section III-B guarantees that any type-correct while program is compiled into a strongly secure RISC program. This is formalized as follows.

Theorem 3 (Strong security enforcement): Let C be a while program, and suppose $\{\}, \epsilon_{lab} \vdash C \hookrightarrow P, \langle w, t\rangle, l, \Phi$. Then P is strongly secure.

According to Theorem 3, we can obtain strongly secure RISC programs from type-correct while programs. Theorem 2 (Section II-C) states that strong security is a sufficient condition to guarantee PoNI. The two results together express a strategy to translate while programs into RISC programs

$$\textbf{seq}\ \frac{\Phi, l \vdash C_1 \hookrightarrow P_1,\ \langle w_1, t_1\rangle,\ l_1,\ \Phi_1 \qquad \Phi_1, l_1 \vdash C_2 \hookrightarrow P_2,\ \langle w_2, t_2\rangle,\ l_2,\ \Phi_2 \qquad t_1 = \mathsf{Trm}\ H \Rightarrow w_2 = \mathsf{Wr}\ H}{\Phi, l \vdash C_1; C_2 \hookrightarrow \left\{\ P_1 +\!\!+ P_2\ \right\},\ \langle w_1 \sqcup w_2, t_1 \uplus t_2\rangle,\ l_2,\ \Phi_2}$$

$$\textbf{while}\ \frac{\begin{array}{c}\lambda = \mathsf{level}(x) = \mathsf{level}(r) \qquad t = \mathsf{Trm}\ H \Rightarrow \mathsf{write}(\lambda) = \mathsf{Wr}\ H \qquad w \sqsubseteq \mathsf{write}(\lambda) \\ \Phi_B \sqsubseteq \Phi[r \leftrightarrow x] \qquad \Phi_B \sqsubseteq \Phi_E[r \leftrightarrow x] \qquad lp, ex\ \text{fresh} \\ \Phi_B, \epsilon_{lab} \vdash C \hookrightarrow P,\ \langle w, t\rangle,\ l',\ \Phi_E \qquad P_i = [\mathsf{load}\ r\ v2p(x),\ \mathsf{store}\ v2p(x)\ r]\end{array}}{\Phi, l \vdash \mathsf{while}\ x\ \mathsf{do}\ C \hookrightarrow \left\{\begin{array}{l} l :\ P_i +\!\!+ [lp : \mathsf{jz}\ ex\ r] +\!\!+ P +\!\!+ \\ l' :\ P_i +\!\!+ [\mathsf{jmp}\ lp]\end{array}\right\},\ \langle \mathsf{write}(\lambda), \mathsf{term}(\lambda) \sqcup t\rangle,\ ex,\ \Phi_B}$$

Fig. 10. Type rules for sequential composition (;) and while

that satisfy PoNI. We state this formally by instantiating the definition of PoNI for RISC programs.

Definition 9 (PoNI for RISC programs): We say that an assembly program P satisfies PoNI if $(P, 0)$ is PoNI according to Definition 6 instantiated on the fault-prone system $\mathcal{A} = \{Loc, \{ch!k | ch \in \{low, high\} \text{ and } k \in \mathbb{W}\} \cup \{\tau\}, \rightarrow\}$.

Corollary 1 (PoNI enforcement on RISC programs): Let C be a while program, and suppose $\{\}, \epsilon_{lab} \vdash C \hookrightarrow P, \langle w, t\rangle, l, \Phi$. Then P is PoNI.

Proof 4: Direct application of Theorem 3 and Theorem 2.

IV. Related Work

Fault Resilient Non-Interference The only previous work of which we are aware that aims to prevent transient faults from violating non-interference is by Del Tedesco et al. [12]. The enforcement approach of that paper is radically different from the approach studied here, and the two approaches are largely complementary. Here we highlight the differences and tradeoffs:

- Targeting a similar RISC machine, the implementation mechanism of [12] is a combination of software fault isolation [33] and a black-box non-interference technique called secure multi-execution [15]. This can be applied to any program, but only preserves the behaviour of noninterfering memory-safe programs. Verifying that a program is memory-safe would have to be done separately, but could be achieved by compiling correctly from a memory-safe language.
- In [12], fault-tolerance is assumed for the code memory but not in the program counter register. The cost of this is that the method described in that paper can only tolerate up to a statically chosen number of faults, whereas in the present work we can tolerate any number.
- The security property enforced by the method described in [12] can be viewed as a restriction of PoNI to runs with a limited number of faults. However, the work does not justify this definition with respect to the more standard notion of probabilistic noninterference. The limitation in the number of faults, together with our result, shows that the established security property is strictly weaker than PNI.

Strong Security for Fault Tolerance Mantel and Sabelfeld [29] used strong security in a state-based encoding of channel-based communication. They observed that strong security is not affected by faults occurring in message transmission. Another way to think of this is that strong security of individual threads implies strong security of their composition; a faulty environment is itself a strongly secure thread, simply because it has no ability to read directly from secrets in the state.

Related Type Systems The type-directed compilation presented here combines several features which are inspired by existing non-interference type systems for sequential and concurrent programming languages. Our security notion is timing sensitive and has some similarities with Agat's [1] type-directed source-to-source transformation method that maps a source program into an equivalent target program where timing leaks are eliminated by padding. Similar ideas were shown to apply to a type system for strong security [30]. Our padding mechanism is different from Agat's, since it is based on counting the number of computation steps in the branches of a high conditional expressions, and our system is more liberal, since it allows e.g. loops with a secret guard. These distinguishing features are both present in Smith's type system for a concurrent language [32] (see also [8]).

Non-interference for Low-level Programming Languages Medel et al. [22] propose a type system for a RISC-like assembly language capable to enforce (termination and time insensitive) non-interference. Enforcing the same security condition, Barthe et al. [5] introduce a stack-based assembly language equipped with a type system. Subsequent work [6] shows a compilation strategy which enforces non-interference across all the intermediate steps until reaching a JVM-like language. Barbuti et al. [4] use a different notion for confidentiality, called σ-security, which is enforced by abstract interpretation.

Dependability The need for a stronger connection between security and dependability has been stated in many works (e.g. [19], [23]). Interestingly, it can be observed that many solutions for dependability are based on information-flow security concepts. In [28], well-known concepts from the information-flow literature are introduced as building blocks to achieve dependability goals. In [35], [20], non-interference-

like definitions are used to express fault tolerance in terms of program semantics.

On the other hand, one could argue that the security domain has been influenced by dependability principles as well. For instance, our enforcement is sound only if fault-tolerant hardware components are deployed for the code memory and the program counter.

Language-Based Techniques for Fault-tolerance The style of our work – in terms of the style of formalisation, the use of programming language techniques, and the level of semantic precision in the stated goals – is in the spirit of Perry et al's fault-tolerant typed assembly language [27]. Because we need to reason about security and not conventional fault tolerance, our semantic model of faults is necessarily much more involved than theirs and more recent variants [18], which are purely nondeterministic.

Security and Transient Faults We have not been the first ones to consider the implications of transient faults for security – Bar-El et al. [3] survey a variety of methods that can be used to induce transient faults on circuits that manipulate sensitive data. Xu et al. [36] study the effect of a single bit flip that strikes the opcode of x86 control flow instructions; their work states the non-modifiability of the source code, which is a crucial assumption in our framework. Bao et al. [2] illustrate several transient-fault based attacks on crypto-schemes. Their protection mechanisms either involve some form of replication or a more intensive usage of randomness (to increase the unpredictability of the result). In a similar scenario, Ciet et al. [11] 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. Canetti et al. [10] 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.

Our approach relies on fault-tolerant support for the program counter. While it seems a bit restrictive, there are fault-tolerant solutions for registers (e.g. [25], [21]).

V. Limitations

The hardware model discussed here is similar to those introduced in [27], [12] and, in common with many informal models of faults, has similar shortcomings: faults occurring at lower levels e.g. in combinatorial circuits, are not modelled. It has been argued [34] that these non-memory elements of a processor have much lower sensitivity to faults than state elements, but in our attacker model this does not say so much.

For timing channels discussed in Section III-B we make a large simplifying assumption: that the time to compute an instruction is constant. In practice modern RISC architectures are not that simple, so there is a need for further refinements to the method to ensure that cache effects are mitigated by preloading or using techniques from [1].

The language we are able to compile is too small to be practical. The minimum required for real examples is to extend to arrays. Our intuition suggests that static arrays and function pointers can be covered in our framework, at the price of deploying more fault-tolerant hardware in the system. Specifically, we believe that array indexing can be secured by deploying an additional dedicated fault-tolerant register, to be used for confining the pointer values within pre-determined address ranges defined at compile time. For function pointers more extensive fault-tolerant hardware would be needed; one could think about a hardened call stack, for guaranteeing that the control flow is not jeopardized by transient faults. Considering the current status of our work, the main challenge in exploring these hypothesis is incorporating them in the already non-trivial correctness proof [13]. In this perspective, a necessary step forward is to move to mechanically verifiable proofs, which will facilitate extending our system to other features, as well as verifying our confidence in the formal results.

The type system presented in Section III is clearly too restrictive. For example, consider a program that leaks secrets through memory operations but does not perform any output action. Clearly, the program fulfils PNI, however it would be rejected by the type system. This is partially explained by the fact that the rules which constitute the type system are meant to enforce a generic timing-sensitive non-interference property, which is not tailored to PNI.

VI. Conclusion

We formalize security in presence of transient faults as Probabilistic Fault-Resilient Non-Interference (PNI). We simplify it by reducing it to a possibilistic framework (PoNI), and we show that another well-known security condition, called Strong Security [30], implies it. We explore a concrete instance of our formalism, a simple RISC architecture for which the only fault-tolerant components are the program counter and the code memory. We define a type system that maps programs written in a simple while-language to the assembly language executed by our architecture and, at the same time, ensures that the produced code satisfies Strong Security (hence PNI).

Acknowledgements

Many thanks to the anonymous referees for useful comments and observations. An earlier version of this work appeared in the first author's PhD thesis, and benefitted from comments from Geoffrey Smith. This work was partially financed by grants from the Swedish research agencies VR and SSF.

REFERENCES

[1] J. Agat, "Transforming out timing leaks," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2000, pp. 40–53.

[2] F. Bao, R. Deng, Y. Han, A. Jeng, A. Narasimhalu, and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *Security Protocols*, ser. Lecture Notes in Computer Science, B. Christianson, B. Crispo, M. Lomas, and M. Roe, Eds. Springer Berlin Heidelberg, 1998, vol. 1361, pp. 115–124. [Online]. Available: http://dx.doi.org/10.1007/BFb0028164

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[4] R. Barbuti, C. Bernardeschi, and N. De Francesco, "Checking security of java bytecode by abstract interpretation," in *Proceedings of the 2002 ACM Symposium on Applied Computing*, ser. SAC '02. New York, NY, USA: ACM, 2002, pp. 229–236. [Online]. Available: http://doi.acm.org/10.1145/508791.508839

[5] G. Barthe, A. Basu, and T. Rezk, "Security types preserving compilation," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds. Springer Berlin Heidelberg, 2004, vol. 2937, pp. 2–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24622-0_2

[6] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference java bytecode verifier," in *Proceedings of the 16th European Conference on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 125–140. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762174.1762189

[7] J. Blmer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (aes)," in *Financial Cryptography*, ser. Lecture Notes in Computer Science, R. Wright, Ed. Springer Berlin Heidelberg, 2003, vol. 2742, pp. 162–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45126-6_12

[8] G. Boudol and I. Castellani, "Non-interference for concurrent programs and thread systems," *Theoretical Computer Science*, vol. 281, no. 1, Jun. 2002.

[9] S. Brookes, "Full abstraction for a shared-variable parallel language," *Information and Computation*, vol. 127, no. 2, pp. 145 – 163, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540196900565

[10] R. Canetti and A. Herzberg, "Maintaining security in the presence of transient faults," in *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '94. London, UK: Springer-Verlag, 1994.

[11] M. Ciet and M. Joye, "Elliptic curve cryptosystems in the presence of permanent and transient faults," *Des. Codes Cryptography*, vol. 36, no. 1, pp. 33–43, Jul. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10623-003-1160-8

[12] F. Del Tedesco, A. Russo, and D. Sands, "Fault tolerant non-interference," in *Proceeding of the International Symposium on Engineering Secure Software and Systems*, ser. LNCS, 2014.

[13] P. Del Tedesco, D. Sands, and A. Russo, "Fault-resilient non-interference (extended version)." [Online]. Available: http://www.cse.chalmers.se/~dave/papers/CSF2016-extended.pdf

[14] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *CACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[15] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. IEEE Computer Society, 2010.

[16] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, ser. SP '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 154–. [Online]. Available: http://dl.acm.org/citation.cfm?id=829515.830563

[17] J. Gray III, "Probabilistic interference," in *Proc. IEEE Symp. on Security and Privacy*, May 1990, pp. 170–179.

[18] R. R. Hansen, K. G. Larsen, M. C. Olesen, and E. R. Wognsen, "Formal methods for modelling and analysis of single-event upsets," in *2015 IEEE International Conference on Information Reuse and Integration, IRI 2015, San Francisco, CA, USA, August 13-15, 2015*, 2015, pp. 287–294. [Online]. Available: http://dx.doi.org/10.1109/IRI.2015.54

[19] E. Jonsson, "Towards an integrated conceptual model of security and dependability," in *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, 2006, pp. 8 pp.–.

[20] G. Lenzini, F. Martinelli, I. Matteucci, and S. Gnesi, "A uniform approach to security and fault-tolerance specification and analysis," pp. 172–201, 2009. [Online]. Available: http://puma.isti.cnr.it/linkdoc.php?icode=2009-A1-013\&authority=cnr.isti\&collection=cnr.isti\&langver=en

[21] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano, "Sparc64 viiifx: A new-generation octocore processor for petascale computing," *Micro, IEEE*, vol. 30, no. 2, pp. 30–40, March 2010.

[22] R. Medel, A. Compagnoni, and E. Bonelli, "A typed assembly language for non-interference," in *Theoretical Computer Science*, ser. Lecture Notes in Computer Science, M. Coppo, E. Lodi, and G. Pinna, Eds. Springer Berlin Heidelberg, 2005, vol. 3701, pp. 360–374. [Online]. Available: http://dx.doi.org/10.1007/11560586_29

[23] D. Nicol, W. Sanders, and K. Trivedi, "Model-based evaluation: from dependability to security," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 48–65, 2004.

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

[25] R. Oliveira, A. Jagirdar, and T. Chakraborty, "A tmr scheme for seu mitigation in scan flip-flops," in *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, March 2007, pp. 905–910.

[26] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-based attack of rsa authentication," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 855–860.

[27] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker, "Fault-tolerant typed assembly language," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 42–53. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250741

[28] J. Rushby, "Partitioning for safety and security: Requirements, mechanisms, and assurance," NASA Langley Research Center, NASA Contractor Report CR-1999-209347, Jun. 1999, also to be issued by the FAA.

[29] A. Sabelfeld and H. Mantel, "Static confidentiality enforcement for distributed programs," in *Static Analysis*, ser. Lecture Notes in Computer Science, M. Hermenegildo and G. Puebla, Eds. Springer Berlin Heidelberg, 2002, vol. 2477, pp. 376–394. [Online]. Available: http://dx.doi.org/10.1007/3-540-45789-5_27

[30] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, ser. CSFW '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 200–. [Online]. Available: http://dl.acm.org/citation.cfm?id=794200.795151

[31] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '02. Springer-Verlag, 2003, pp. 2–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=648255.752727

[32] G. Smith, "A new type system for secure information flow," in *Proc. IEEE Computer Sec. Foundations Workshop*, Jun. 2001.

[33] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/168619.168635

[34] 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 *International Conference on Dependable Systems and Networks (DSN 2004)*, 2004.

[35] D. G. Weber, "Formal specification of fault-tolerance and its relation to computer security," in *Proceedings of the 5th international workshop on Software specification and design*, ser. IWSSD '89. New York, NY, USA: ACM, 1989, pp. 273–277. [Online]. Available: http://doi.acm.org/10.1145/75199.75240

[36] J. Xu, S. Chen, Z. Kalbarczyk, and R. Iyer, "An experimental study of security vulnerabilities caused by errors," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, 2001, pp. 421–430.

APPENDIX A
DETAILS AND PROOFS OF MAIN RESULTS (§II)

A. Proof of Proposition 1

In order to prove Proposition 1, we prove an auxiliary result which ensures that our assignment of probability to n-sized runs is a probability distribution.

Proposition 2: Let Sys be a fault-prone system and Env an environment. Then $\forall Z \in Sys \times Env$ and $\forall n \geq 0$ $\Sigma_{r \in \mathsf{run}_n(Z)} \mathsf{Pr}(r) = 1$.

Proof 5: We prove the statement by induction on n.

Base case

When $n = 0$, the set $\mathsf{run}_0(Z)$ contains only one element, the empty run, and its probability is 1.

Inductive step

Consider $n > 0$.

Any run $r \in \mathsf{run}_n(Z)$ can be written as $r = Z \xrightarrow{a_0}_{p_0} Z_1 \dots Z_{n-1} \xrightarrow{a_{n-1}}_{p_{n-1}} Z_n = (Z \xrightarrow{a_0}_{p_0} Z_1 \dots Z_{n-1}).(Z_{n-1} \xrightarrow{a_{n-1}}_{p_{n-1}} Z_n)$, where $r' = Z \xrightarrow{a_0}_{p_0} Z_1 \dots Z_{n-1}$ is a prefix of r in $\mathsf{run}_{n-1}(Z)$ with probability $\mathsf{Pr}(r') = p_{r'}$.

Consider the set $R_{r'} \subseteq \mathsf{run}_n(Z)$ of all n-sized runs from Z that has r' as prefix. Since for all subsets in $\wp(F)$ there is a transition rule in $Sys \times Env$, we have $\Sigma_{r \in \mathsf{run}_1(Z_{n-1})} \mathsf{Pr}(r) = 1$. Hence $\Sigma_{r \in R_{r'}} \mathsf{Pr}(r) = p_{r'}$.

Hence we conclude that $\Sigma_{r \in \mathsf{run}_n(Z)} \mathsf{Pr}(r) = \Sigma_{r' \in \mathsf{run}_{n-1}(Z)} \Sigma_{r \in R_{r'}} \mathsf{Pr}_Z(r) = \Sigma_{r' \in \mathsf{run}_{n-1}(Z)} p_{r'} = 1$, where the last result holds by inductive hypothesis.

Proof 6 (Proof of Proposition 1): Recall that for any trace $t \in (LAct \cup \{\tau\})^n$, $\mathsf{Pr}_Z(t) = \Sigma_{\{r \in \mathsf{run}_n(Z) | \mathsf{trace}(r) = t\}} \mathsf{Pr}(r)$. Since for any run r there is a trace $t \in (LAct \cup \{\tau\})^n$ such that $\mathsf{trace}(r) = t$, we have as a result that $\Sigma_{t \in (LAct \cup \{\tau\})^n} \mathsf{Pr}_Z(t) = \Sigma_{r \in \mathsf{run}_n(Z)} \mathsf{Pr}(r) = 1$, where the last equality holds because of Proposition 2.

B. Full definitions of Augmented Fault-prone and Termination Transparent Systems

An augmented fault-prone system is formally described as follows.

Definition 10 (Augmented Fault-prone System): Given a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$ we define the augmented system Sys^+ as $Sys^+ = \{Loc, Act \times \wp(F), \rightsquigarrow\}$ by the following rules:

$$\frac{\exists l S \xrightarrow{l} S' \quad \mathsf{flip}(S, L) \xrightarrow{a} S' \quad L \subseteq F}{S \overset{L,a}{\rightsquigarrow} S'}$$

$$\frac{\exists l S \xrightarrow{l} S' \quad \mathsf{flip}(S, L) \not\rightarrow \quad L \subseteq F}{S \overset{L,\tau}{\rightsquigarrow} \mathsf{flip}(S, L)}$$

$$\frac{S \not\rightarrow \quad L \subseteq F}{S \overset{L,\tau}{\rightsquigarrow} S}$$

A termination transparent system is formalized as follows.

Definition 11 (Termination Transparent System): For a fault-prone system $Sys = \{Loc, Act, \rightarrow\}$ we define its termination-transparent version as $Sys^\infty = \{Loc, Act, \rightarrow_\infty\}$ where $\rightarrow_\infty$ is defined with the following rules:

$$\frac{S \xrightarrow{a} S'}{S \xrightarrow{a}_\infty S'} \qquad \frac{S \not\rightarrow}{S \xrightarrow{\tau}_\infty S}$$

C. Proof of Theorem 1

Before proving the theorem in question, we need to define some auxiliary concepts.

Definition 12 (Enabling set): Let $Z = \langle\ S,\ E\ \rangle$ and $Z' = \langle\ S',\ E'\ \rangle$ be a pair of states in $Sys \times Env$ such that $Z \xrightarrow{a}_p Z'$. We say that L is an enabling set (of locations) for $Z \xrightarrow{a}_p Z'$ in the following cases:

- the transition is derived from rule $[\mathbf{Step}]$ and $L \in \pi$;
- the transition is derived from rule $[\mathbf{Stuck-1}]$ and L is the argument in $\mathsf{flip}(S, L)$;
- the transition is derived from rule $[\mathbf{Stuck-2}]$.

Definition 13 (Enabling sequence): Let r be a run for $\langle\ S_0,\ E_0\ \rangle$ in $Sys \times Env$ such that $\langle\ S_0,\ E_0\ \rangle \xrightarrow{a_0}_{p_0} \langle\ S_1,\ E_1\ \rangle \dots \xrightarrow{a_{n-1}}_{p_{n-1}} \langle\ S_n,\ E_n\ \rangle$. The sequence $\mathcal{L} = L_0 \dots L_{n-1}$ such that $\forall 0 \leq i \leq n-1$ L_i is an enabling set for $\langle\ S_i,\ E_i\ \rangle \xrightarrow{a_i}_{p_i} \langle\ S_{i+1},\ E_{i+1}\ \rangle$ is called an enabling sequence for r. We define the probability of $\mathcal{L}$ as $\mathsf{Pr}_r(\mathcal{L}) = \Pi_{0 \leq i \leq n-1} \mathsf{Fault}(E_i)(L_i)$. We define the set of all enabling sequences for r as $\phi(r)$.

We also need the following intermediate result.

Lemma 1: Let r be a run for Z in $Sys \times Env$. Then we have that $\mathsf{Pr}(r) = \Sigma_{\mathcal{L} \in \phi(r)} \mathsf{Pr}_Z(\mathcal{L})$.

We can now prove Theorem 1.

Proof 7 (Proof of Theorem 1): Suppose P enjoys PoNI, we now show it enjoys PNI as well.

Consider a faulty system Sys, an error environment $Env = (Err, \mathsf{Fault})$ and two states $Z = \langle\ S,\ E\ \rangle$ and $Z' = \langle\ S',\ E\ \rangle$, for $S, S' \in Sys$ and $E \in Err$. Assume $S\big|_{ProgLoc} = S'\big|_{ProgLoc} = P$ and $S =_L S'$.

We first show that for any $n \geq 0$ and for any trace $t \in (LAct \cup \{\tau\})^n$, $\mathsf{Pr}_Z(t) \leq \mathsf{Pr}_{Z'}(t)$, and hence by symmetry that $\mathsf{Pr}_Z(t) = \mathsf{Pr}_{Z'}(t)$.

We prove the inequality by relating the probability of a trace to the probability determined by the enabling sequences that corresponds to it.

Consider a trace t such that $\mathsf{Pr}_Z(t) > 0$. Let $\rho_Z(t)$ defined as $\rho_Z(t) = \{r \in \mathsf{run}(Z) | \mathsf{trace}(r) = t\}$ be the (nonempty) set of runs from Z whose trace is t.

We have $\mathsf{Pr}_Z(t) = \Sigma_{r \in \rho_Z(t)} \mathsf{Pr}(r) = \Sigma_{r \in \rho_Z(t)} \Sigma_{\mathcal{L} \in \phi(r)} \mathsf{Pr}_Z(\mathcal{L})$, where the first equality holds by definition and the second one follows from Lemma 1.

We now show that all enabling sequences for Z are also enabling sequences for Z'. Let $\kappa_Z = \cup_{r \in \rho_Z(t)} \phi(r)$ be the set of all enabling sequences for t in Z and let $\mathcal{L}$ be an enabling sequence for a run $r \in \mathsf{run}(Z)$. Since P is PoNI, there must be a run r' from Z' such that $\mathcal{L}$ is an enabling sequence for r', and $\mathsf{trace}(r') = t$. Hence, for the set $\kappa_{Z'} = \cup_{r' \in \rho_{Z'}(t)} \phi(r')$ we have that $\kappa_Z \subseteq \kappa_{Z'}$.

Also, observes that for any $\mathcal{L} \in \kappa_Z$. $\mathsf{Pr}_Z(\mathcal{L}) = \mathsf{Pr}_{Z'}(\mathcal{L})$, since $\mathsf{trace}(u) = \mathsf{trace}(u')$.
Then $\mathsf{Pr}_Z(t) = \Sigma_{\mathcal{L} \in \kappa_Z} \mathsf{Pr}_Z(\mathcal{L}) \leq \Sigma_{\mathcal{L} \in \kappa_{Z'}} \mathsf{Pr}_{Z'}(\mathcal{L}) = \mathsf{Pr}_{Z'}(t)$.
We continue by showing that PNI implies PoNI by proving the contrapositive. Suppose that P is not PoNI. Then there must a fault-prone system Sys, two states S, S' such that $S\big|_{ProgLoc} = S'\big|_{ProgLoc} = P$ and $S =_L S'$, together with a location set $\lambda \in \wp(F)$, a trace $t = L_0, a_0, \ldots, L_j, a_j$ and $a, b \in LAct \cup \{\tau\}$ such that $a \neq b$, $S \overset{t,\lambda,a}{\rightsquigarrow}$ and $S' \overset{t,\lambda,b}{\rightsquigarrow}$.
Define an error environment $Env = (Err, \mathsf{Fault})$ such that $Err = \langle \{L_i | L_i \in t\} \cup \{\lambda\}, LAct \cup \{\tau\}, \{L_i \overset{a}{\rightsquigarrow} L_{i+1} | 0 \leq i \leq j-1 \text{ and } a \in LAct \cup \{\tau\}\} \cup \{L_j \overset{a}{\rightsquigarrow} \lambda | a \in LAct \cup \{\tau\}\}\rangle$ and $\mathsf{Fault}(\lambda)(\lambda) = \mathsf{Fault}(L)(L) = 1$. Essentially, Env deterministically traverses all flipped locations included in t, and terminates in λ, regardless of the actions performed by the fault-prone system.
Consider now the composition of Sys with Env and let $Z = \langle\ S,\ L_0\ \rangle$ and $Z' = \langle\ S',\ L_0\ \rangle$. Let $t' = a_0 \ldots . a_j . a$ be a trace in $(LAct \cup \{\tau\})^*$ obtained from t by (i) striping flipped locations and (ii) appending the action a at the end. Then there exists a unique run $r \in \mathsf{run}(Z)$ such that $\mathsf{trace}(r) = t'$ and $\mathsf{Pr}(r) = \mathsf{Pr}_Z(t') = 1 \neq \mathsf{Pr}_{Z'}(t') = 0$. The inequality between $\mathsf{Pr}_Z(t')$ and $\mathsf{Pr}_{Z'}(t')$ follows from the hypothesis of P not being PoNI, which implies that there is no $r' \in \mathsf{run}(Z')$ such that $\mathsf{trace}(r') = t'$.

D. Proof of Theorem 2

Rather than showing that Strong Security implies PoNI directly, we take an indirect approach.

First we characterize the semantics of a termination-transparent system in terms of "transition traces", borrowing ideas from [9]. Then we define an ad-hoc security property, called Strong Trace-based Security within this semantic model. We finally show that Strong Security implies Strong Trace-based Security, which in turn implies PoNI.

For improving readability we represent $S\big|_{LowLoc \cup HighLoc}$ as M, the data component, therefore the state of a fault-prone system is represented as (P, M). We adapt the concept of low equality between states to data components by saying that $M =_L M'$ if $M\big|_{LowLoc} = M'\big|_{LowLoc}$.

Definition 14 (Transition trace semantics): Let Sys be a fault-prone system and $Sys^\infty = \{Loc, Act, \rightarrow_\infty\}$ be its termination-transparent version. The n-step transition-trace semantic of a program component P_0 is defined as $\mathcal{T}_n(P_0) = \{(M_0, a_0, M_0'), (M_1, a_1, M_1') \ldots (M_{n-1}, a_{n-1}, M_{n-1}') | \ \forall 0 \leq i \leq n-1 (P_i, M_i) \xrightarrow{a_i}_\infty (P_{i+1}, M_i')\}$. The transition trace semantics of P_0 is defined as $\mathcal{T}(P_0) = \cup_n \mathcal{T}_n(P_0)$.

In the transition trace model, the semantics of a program component P is built in sequences of steps. In particular, at any step, the program component is executed on a certain data component, then the data component is modified and the execution is restarted. Observe that the model is very similar to the way a fault-prone system and an error environment interact with each other. This is even more clear when viewing the modification of the data component as the effect of its interaction with the error environment.

We say that two transition traces

$$t = (M_0, a_0, M_0'), (M_1, a_1, M_1') \ldots (M_{n-1}, a_{n-1}, M_{n-1}')$$
$$t' = (N_0, b_0, N_0'), (N_1, b_1, N_1') \ldots (N_{n-1}, b_{n-1}, N_{n-1}')$$

are input low-equivalent, written $t =_I t'$, if $\forall 0 \leq i < n$, $M_i =_L N_i$, whereas they are output low-equivalent, written $t =_O t'$ if $\forall 0 \leq i < n$, $low(a_i) = low(b_i)$ and $M_i' =_L N_i'$.

Definition 15 (Strong Trace-based Security (StbS)): We say that a program component P is n-Strong Trace-based Secure if for any two transition traces $t, t' \in \mathcal{T}_n(P)$, if $t =_I t'$ then $t =_O t'$. We say that a program component P is Strong Trace-based Secure if it is n-Strong Trace-based Secure for any $n \in \mathbb{N}$.

We now show how to use the notion of Strong Trace-based Security to bridge the gap between Strong Security and PoNI. We show that Strong Security implies Strong Trace-based Security first.

Lemma 2 (SS implies StbS): Let P be a program component. If P enjoys SS then P enjoys StbS.

Proof 8: We define some notation first. We refer to the i-th triple in a transition trace t as t_i, and to the program component used to evaluate it as P_{t_i} (for a trace $t = (M_0, a_0, M_0'), (M_1, a_1, M_1') \ldots (M_{n-1}, a_{n-1}, M_{n-1}')$ we therefore say that the i-th triple (M_i, a_i, M_i') is induced by $(P_{t_i}, M_i) \xrightarrow{a_i}_\infty (P_{t_{i+1}}, M_i')$).

Consider a program component P and two n-transition traces

$$t = (M_0, a_0, M_0'), (M_1, a_1, M_1') \ldots (M_{n-1}, a_{n-1}, M_{n-1}')$$

and

$$t' = (N_0, b_0, N_0'), (N_1, b_1, N_1') \ldots (N_{n-1}, b_{n-1}, N_{n-1}')$$

in $\mathcal{T}_n(P)$.

We want to show that if P enjoys SS and $t =_I t'$, then $t =_O t'$.

Starting from a strong bisimulation R for (P, P), the idea of the proof is to infer properties of t' by unwinding R for n-steps. We proceed by showing that for all $0 \leq i < n$ we have that $(P_{t_i}, P_{t_i'}) \in R$. For $i = 0$ $P_{t_0} = P_{t_0'} = P$ and $(P, P) \in R$. If $(P_{t_i}, P_{t_i'}) \in R$, then by definition of R we have that if $(P_{t_i}, M_i) \xrightarrow{a_i}_\infty (P_{t_{i+1}}, M_i')$ and $M_i =_L N_i$ then $(P_{t_i'}, N_i) \xrightarrow{t_i}_\infty (P_{t_{i+1}'}, N_i')$ and $(P_{t_{i+1}}, P_{t_{i+1}'}) \in R$. But this is the case for t and t', since $t =_I t'$.

The statement of the lemma is therefore proved by recalling that two program components P and P' in a strong bisimulation R are such that their executions from low equivalent data result in (i) low equivalent data and (ii) low equivalent actions.

We now discuss the relation between Strong Trace-based Security and PoNI. In general it is not true that Strong Trace-based Security is stronger than PoNI. Consider, for example, the class of systems such that $ProgLoc \not\subseteq T$. Due to transient faults, a completely innocuous program component can be

converted into a harmful one, even when it enjoys Strong Trace-based Security.

Surprisingly, this is not the only constraint that we must impose to the systems of our interest. We must also require that they show a uniform behavior for termination, as shown in the following example.

Example 2: Consider the fault-prone system in Figure 11. For each state $S = \{b_i \to \{0,1\} | i \in \{0,1,2\}\}$ we consider $ProgLoc = \{b_0\}$, $HighLoc = \{b_1\}$ and $LowLoc = \{b_2\}$. We also assume that the states where P is 1 are stuck and therefore are omitted.

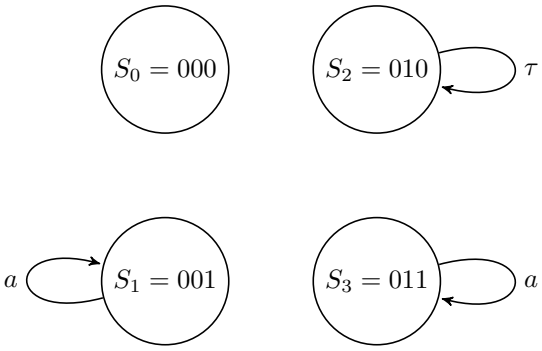

Fig. 11. StbS does not imply PoNI in general

The system is Strongly Trace-based Secure: all states are either stuck or perform a transition on themselves, therefore low equivalence is preserved. The only difference in the output behavior is observable between S_0 and S_2: the former is stuck, the latter perform a τ transition. Nonetheless they result indistinguishable in the termination transparent version of the system. However, the system is not PoNI. In fact, if a bit flip on b_2 can transform S_2 into S_3 so we have $S_0 \overset{\{b_2\},\tau}{\rightsquigarrow}$ but $S_2 \overset{\{b_2\},a}{\not\rightsquigarrow}$.

From now onwards we focus our attention on "standard fault-prone" systems. For such systems, we have that the whole program component is fault-tolerant (formally $ProgLoc \subseteq T$) and it is either stuck or active, regardless of the data component.

Definition 16 (Standard fault-prone systems): A fault-prone system is called standard if $ProgLoc \subseteq T$ and, for all P, either for any data component M there exists an action l such that $(P, M) \xrightarrow{l}$ or for any data component M the system is stuck, namely $(P, M) \not\to$.

For the class of systems of our interest, Strong Trace-based Security is indeed stronger than PoNI.

Lemma 3 (Strong Trace-based Security implies PoNI): Let P be a program component in a standard fault-prone system Sys. If P enjoys StbS then P enjoys PoNI.

Proof 9:

We prove this lemma by showing the contrapositive. Suppose that P is not PoNI. Then there must be two states $S = (P, M_0)$ and $S' = (P, N_0)$ in the augmented version of a fault-prone system Sys such that $M_0 =_L N_0$, and two runs that exit from them whose corresponding traces violate the security condition. Let

$$r = (P, M_0) \overset{L_0,a_0}{\rightsquigarrow} (P_1, M_1) \overset{L_1,a_1}{\rightsquigarrow} \ldots (P_{j-1}, M_{j-1}) \overset{L_{j-1},a_{j-1}}{\rightsquigarrow} (P_j, M_j) \overset{L,a}{\rightsquigarrow}$$

and

$$r' = (P, N_0) \overset{L_0,b_0}{\rightsquigarrow} (P^1, N_1) \overset{L_1,b_1}{\rightsquigarrow} \ldots (P^{j-1}, N_{j-1}) \overset{L_{j-1},b_{j-1}}{\rightsquigarrow} (P^j, N_j) \overset{L,b}{\rightsquigarrow}$$

be the runs in question, such that $\forall 0 \leq i < j$ $low(a_i) = low(b_i)$ but $low(a) \neq low(b)$.

Before continuing, we observe that, in the initial configuration, P cannot be stuck. Also, it must be that at most one run between r and r' contains a sequence of stuck configurations. Both conditions are necessary to have $low(a) \neq low(b)$.

We now show that it is possible to build two transition traces for P that violate Strong Trace-based Security. Recall that the flip function can be applied only to locations in F, and that we consider systems whose faulty locations are restricted to the data component ($F \subseteq LowLoc \cup HighLoc$). Hence, when $S = (P, M)$, we write $\mathsf{flip}(S, L)$ as $(P, \mathsf{flip}(M, L))$, and we focus on the data component $\mathsf{flip}(M, L)$ when necessary.

We proceed by distinguishing two cases, depending on whether or not a stuck configuration is traversed by r (equivalently r').

Case 1: no stuck configurations are traversed in either r or r'.

Consider the following two transition traces

$$t = (E_0, a_0, M_1), (E_1, a_1, M_2) \ldots (E_j, a, M_{j+1})$$

and

$$t' = (F_0, b_0, N_1), (F_1, b_1, N_2) \ldots (F_j, b, N_{j+1})$$

where $E_i = \mathsf{flip}(M_i, L_i)$, $F_i = \mathsf{flip}(N_i, L_i)$ for some $\{M_i\}_{i \in \{1 \ldots j\}}, \{N_i\}_{i \in \{1 \ldots j\}}$, and where $L_j = L$. Since r does not traverse stuck configurations, all transitions are computed by an application of the rule [**Step**]. This means that for any transition we have $(P_i, M_i) \overset{L_i,a_i}{\rightsquigarrow} (P_{i+1}, M_{i+1})$ if $(P_i, \mathsf{flip}(M_i, L_i)) \xrightarrow{a_i} (P_{i+1}, M_{i+1})$. Hence t is in $\mathcal{T}(P)$. By applying a similar argument for r', we conclude that t' is in $\mathcal{T}(P)$ as well.

Observe that flip preserves low equivalence between data components (if $M_i =_L N_i$ then for all set of locations L it is true that $\mathsf{flip}(M_i, L) =_L \mathsf{flip}(N_i, L)$). Considering that $M_0 =_L N_0$, there are two possible cases. Either there exists k, such that $0 \leq k < j$ and $(\mathsf{flip}(M_k, L_k), a_k, M_{k+1})$ and $(\mathsf{flip}(N_k, L_k), b_k, N_{k+1})$ and $M_{k+1} \neq_L N_{k+1}$, or $\forall k 0 \leq k \leq j$ $M_k =_L N_k$ and $low(a) \neq low(b)$. In both cases Strong Trace-based Security is violated.

Case 2: there is a stuck configuration in r.

We consider the case in which a configuration in r is stuck. The symmetric case for r' is similar, and it is omitted.

Since Sys is a "standard fault-prone" system, the rule [$\mathbf{Stuck-2}$] cannot be applied in the first step. Let $1 \leq w \leq j$

be the index of the first stuck state (P_w, M_w) in r. Consider the following transition traces

$$\begin{aligned} t &= (E_0, a_0, M_1), \ldots, (E_{w-1}, a_w, M_w), \\ &\quad (E_w, \tau, E_w), \ldots, (E_j, \tau, E_j) \\ t' &= (F_0, b_0, N_1), \ldots, (F_{w-1}, b_{w-1}, N_w), \\ &\quad (F_w, b_w, N_{w+1}), \ldots, (F_j, b, N_{j+1}) \end{aligned}$$

where $E_i = \mathsf{flip}(M_i, L_i)$, $F_i = \mathsf{flip}(N_i, L_i)$ for some $\{M_i\}_{i\in\{1\ldots j\}}, \{N_i\}_{i\in\{1\ldots j\}}$, and where $L_j = L$.

As observed in the previous case, since flip preserves low equivalence of data components, there are the following cases to be considered. Either there exists k such that $0 \leq k < w$ and $(\mathsf{flip}(M_k, L_k), a_k, M_{k+1})$ and $(\mathsf{flip}(N_k, L_k), b_k, N_{k+1})$ and $M_{k+1} \neq_L N_{k+1}$, or there exists k such that $w \leq k < j$ and $(\mathsf{flip}(N_k, L_k), \tau, \mathsf{flip}(N_k, L_k))$ and $(\mathsf{flip}(N_k, L_k), b_k, N_{k+1})$ and $\mathsf{flip}(N_k, L_k) \neq_L N_{k+1}$, or $\tau \neq low(b)$. In all cases Strong Trace-based Security is violated.

Proof 10 (Proof of Theorem 2): Directly obtained by applying Lemma 2 and Lemma 3.