Detecting Fault Injection Attacks with Runtime Verification
Ali Kassem, Yliès Falcone

To cite this version:

HAL Id: hal-02283434
https://hal.archives-ouvertes.fr/hal-02283434
Submitted on 1 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Fault injections are increasingly used to attack/test secure applications. In this paper, we define formal models of runtime monitors that can detect fault injections that result in test inversion attacks and arbitrary jumps in the control flow. Runtime verification monitors offer several advantages. The code implementing a monitor is small compared to the entire application code. Monitors have a formal semantics; and we prove that they effectively detect attacks. Each monitor is a module dedicated to detecting an attack and can be deployed as needed to secure the application. A monitor can run separately from the application or it can be “weaved” inside the application. Our monitors have been validated by detecting simulated attacks on a program that verifies a user PIN.

Keywords Runtime Verification, Monitor, Fault Injection, Detection, Quantified Event Automata, Attacker Model

1 Introduction

Fault injections are effective techniques to exploit vulnerabilities in embedded applications and implementations of cryptographic primitives [1, 2, 3, 4, 5, 6]. Fault injections can be thwarted (or detected) using software or hardware countermeasures [7, 8]. Hardware countermeasures are expensive and unpractical for off-the-shelf products. Henceforth, software countermeasures are commonly adopted. Software countermeasures can be categorized into algorithm-level and instruction-level countermeasures.

Most of the existing algorithm-level countermeasures make use of redundancy such as computing a cryptographic operation twice then comparing the outputs [9], or using parity bits [10]. Algorithm-level countermeasures are easy to implement and deploy and usually introduce low (computational and footprint) overhead. However, as they are usually based on redundancy, they can be broken using multiple faults injection or by skipping critical instruction [11, 12].

In the other hand, instruction-level countermeasures require changes to the low level instruction code, for example by applying instruction duplication or triplication [13, 14]. Instruction-level countermeasures are more robust than algorithm-level countermeasures. Indeed, it is believed that instruction-level countermeasures are secure against multiple faults injection under the assumption that skipping two consecutive instructions is too expensive and requires high synchronization capabilities [13, 15]. However, instruction-level countermeasures introduce a large overhead, for instance, instruction duplication doubles the execution time. Moreover, instruction-level countermeasures require changing the instruction set (with dedicated compilers) since e.g., some instructions have to be replaced by a sequence of idempotent or semantically equivalent instructions.

In this paper, we use runtime verification principles [16, 17, 18, 19] and monitors to detect fault injections that result in test inversion or unexpected jumps in the control flow of a program. A monitor can run off-line after the end of an execution provided that the necessary events have been saved in a safe log, or on-line in parallel with the execution.
We use Quantified Event Automata (QEAs)\cite{20} to express our monitors. We prove that our monitors for test inversion and jump attacks detect them if and only if such attack occur at runtime -- Propositions 1 and 2, respectively. From an implementation point of view, we validate Java implementations of our monitors using attack examples on a program that verifies a user PIN code taken from the FISSC benchmark\cite{21}. Our monitors are lightweight and small in size. The monitors execution time is proportional to the size of the program under surveillance. However, the memory overhead can be bounded as only data related to the “active” basic block need to be kept in memory.

This make them suitable for low-resource devices where monitors can run in a small (hardware)-secured memory (where the entire application may not fit). Moreover, monitors can run separately from the application under surveillance, and thus can run remotely (in a secure environment) provided a secure communication channel.

The rest of the paper is structured as follows. Section 2 overviews Quantified Event Automata. Section 3 introduces preliminaries. Section 4 introduces execution and attacker models. Section 5 introduces our monitors. Section 6 describes an experiment that validates the effectiveness of our monitors. Section 7 discusses related work. Finally, Section 8 concludes and outlines avenues for future work.

## 2 Quantified Event Automata

We briefly overview Quantified Event Automata\cite{20} (QEAs) which are used to express monitors. QEAs are an expressive formalism to represent parametric specifications to be checked at runtime. An Event Automaton (EA) is a (possibly non-deterministic) finite-state automaton whose alphabet consists of parametric events and whose transitions may be labeled with guards and assignments. The syntax of an EA is built from a set of event names $\mathcal{N}$, a set of values $\mathcal{Val}$, and a set of variables $\mathcal{Var}$ (disjoint from $\mathcal{Val}$). The set of symbols is defined as $\mathcal{Sym} = \mathcal{Val} \cup \mathcal{Var}$. An event is a tuple $(\alpha, p_1, \ldots, p_n)$, where $\alpha \in \mathcal{N}$ is the event name and $p_1, \ldots, p_n \in \mathcal{Sym}^n$ are the event parameters. We use a functional notation to denote events: $(\alpha, p_1, \ldots, p_n)$ is denoted by $\alpha(p_1, \ldots, p_n)$. Events that are variable-free are called ground events, i.e., an event $\alpha(p_1, \ldots, p_n)$ is ground if $p_1, \ldots, p_n \in \mathcal{Val}^n$. A trace is defined as a finite sequence of ground events. We denote the empty trace by $\epsilon$.

The semantics of an EA is close to the one of a finite-state automaton with the natural addition of guards and assignments on transitions. A transition can be triggered only if its guard evaluates to True with the current binding (a map from variables to concrete values), and the assignment updates the current binding.

A QEA is an EA with some (or none) of its variables quantified by $\forall$ or $\exists$. Unquantified variables are left free and can be manipulated through assignments and updated during the processing of the trace. A QEA accepts a trace if after instantiating the quantified variables with the values derived from the trace, the resulting EAs accept the trace. Each EA consumes only a certain set of events, however a trace can contain other events which are filtered out. The quantification $\forall$ means that a trace has to be accepted by all EAs, while the quantification $\exists$ means that it has to be accepted by at least one EA. For a QEA $\mathcal{M}$ with quantified variables $x_1, \ldots, x_n$. We use the functional notation $\mathcal{M}(x_1, \ldots, x_n)$ to refer to the related EAs depending on the values taken by $x_1, \ldots, x_n$.

We depict QEAs graphically. The initial state of a QEA has an arrow pointing to it. The shaded states are final states (i.e., accepting states), while white states are failure states (i.e., non-accepting states). Square states are closed-to-self (aka skip-states), i.e., if no transition can be taken, then there is an implicit self-looping transition. We use the notation $\exists_{\text{guard}}$ assignment to write guards and assignments on transitions, $:= \text{for variable assignment, and} \equiv \text{for equality test.}$

**Example 1 (QEA).** Figure 1 depicts $\mathcal{M}_R$, a QEA that checks whether a given trace satisfies requirement $R$: “for all $i$, event $e_2(i)$ should precede event $e_2(i)$”. The alphabet of $\mathcal{M}_R$ is $\Sigma_R = \{e_2(1), e_2(\bot)\}$. Consequently, any event in an input trace that is not an instantiation of $e_2(1)$ or $e_2(\bot)$ is ignored by $\mathcal{M}_R$. QEA $\mathcal{M}_R$ has two states (1) and (2), which are both accepting states, one quantified variable $i$, and zero free variables. As its initial state (1) is an accepting state, the empty trace is accepted by $\mathcal{M}_R$. State (1) is a square state, hence an event $e_2(1)$ at state (1) leads into an implicit failure state, which is what we want as it would not be preceded by event $e_2(1)$ in this case. An event $e_2(\bot)$ at state (1) leads into state (2) which is a skipping state, so after an occurrence of event $e_2(1)$ any sequence of events (for the
## 3 Preliminaries and Notations

As a running example, we consider function `verifyPIN` which is depicted in Listing 1\(^1\). Function `verifyPIN` is the main function for the verification of a user PIN. It handles the counter of user trials (variable `g_ptc`), which is initialized to 3, i.e., the user is allowed for 3 trials (one trial per execution). The user is authenticated if the value of `g_ptc` is greater than 0, and function `byteArrayCompare` returns `BOOL_TRUE`. Function `byteArrayCompare` returns `BOOL_TRUE` if `g_userPin` and `g_cardPin` are equal. Note that `BOOL_TRUE` and `BOOL_FALSE` have the values 0xAA and 0x55, respectively. This provides a better protection against faults that modifies data-bytes.

The monitors defined in Section 5 are generic. They are independent from programming language and do not require changes to the low-level code because the required instrumentation to produce events can be made at the the source code level. However, to describe attacks at a lower level, we make use of the three-address code (TAC) representation. TAC is an intermediate-code representation which reassembles for instance LLVM-IR. TAC is machine independent, easy to generate from source code, and can be easily converted into assembly code. In TAC, a program is a finite sequence of three-address instructions. In particular, an instruction `ifZ z goto L` is a conditional branch instruction that directs the execution flow to L if the value of z is 0 (i.e., false). An instruction `goto L` is an unconditional branch instruction that directs the execution flow to L. A label L can be assigned to any instruction in the TAC. Instruction `Push x` pushes the value of x onto the stack. Before making a function call, parameters are individually pushed onto the stack from right to left. While `Pop k` pops k bytes from the stack; it is used to pop parameters after a function call.

### Example 2. Listing 2 depicts the TAC representation of `verifyPIN`. The variables _t0, _t1 and _t2 are compiler-generated temporaries.

To specify how the program under verification has to be instrumented, in Section 4.1, we refer to the control flow graph (CFG) of the program. The CFG of a program is a representation of all the paths that might be taken during its execution. A CFG is a rooted directed graph \((V, E)\), where \(V\) is a set of nodes representing basic blocks, and \(E\) is a set of directed edges representing possible control flow paths between basic blocks. A CFG has an entry node and one exit node. The entry node has no incoming edges while the exit node has no outgoing edges. A basic block is a maximal sequence \(S_1 \ldots S_n\) of instructions such that

\(^1\) The code is inspired from the C version in the FISSC benchmark [21].
Detecting Fault Injection Attacks with Runtime Verification

![CFG diagram for the verifyPIN function.]

- it can be entered only at the beginning, i.e., none of the instructions $S_2 \ldots S_n$ has a label (i.e., target of a branch), and
- it can be exited only at the end, i.e., none of the instructions $S_1 \ldots S_{n-1}$ is a branch instruction or a return.

A CFG may contain loops. A loop is a sequence $B_1, \ldots, B_n$ of basic blocks dominated by the first basic block: $B_1$, and having exactly one back-edge from the last basic block: $B_n$ into $B_1$. Note that in a CFG, a basic block $B$ dominates a basic block $B'$ if every path from the entry node to $B'$ goes through $B$. An edge $(B', B)$ is called a back-edge if $B$ dominates $B'$.

Example 3. Figure 2 depicts the CFG of verifyPIN.

We define a test, in TAC, as an instruction that involves a logical expression, i.e., an instruction of the form $z := (x \text{ oprel } y)$ where `oprel` is a logical operator, followed by the conditional branch instruction `ifZ z goto L` for some label $L$. For a test $A$ we use the notation `cond(A)` to refer to the condition (i.e., the logical expression) involved in $A$.

Listing 2: The TAC representation of verifyPIN.

```
verifyPIN:
g_authenticated := BOOL_FALSE
t0 := (g_ptc > 0)
ifZ t0 goto L1
Push g_cardPin
Push g_userPin
t1:= call byteArrayCompare
Pop 64
t2 := (_t1 == BOOL_TRUE)
ifZ t2 goto L2

B3:
g_ptc := 3
g_authenticated := BOOL_TRUE
goto L1

L2: g_ptc := -g_ptc
```

Figure 2: CFG for the verifyPIN function.
Example 4 (Test). Function \texttt{verifyPIN} contains two tests:

- \_t0 := (g\_ptc > 0), \_fZ \_t0 \_goto L1
- \_t2 := (\_t1 == BOOL\_TRUE), \_fZ \_t2 \_goto L2.

Let \( B \) be a basic block. We use the notation \( B_L \) to refer to the successor of \( B \) whose head is labeled by \( L \) if the tail of \( B \) is an unconditional jump instruction \_goto L. Similarly, we use the notation \( B_T \) (resp. \( B_F \)) to refer to the successor of \( B \) that is executed when \((x \ oprel y) = \text{True} \) (resp. \((x \ oprel y) = \text{False}\)) if \( B \) ends with a test \( A \) where \( \text{cond}(A) = (x \ oprel y) \). Note that \( B_F \) is the basic block whose head is labeled by \( L \).

Finally, in what follows, variable \( i \) is used to denote the unique identifier of a basic block \( B \), and \( i_{\square} \) is used to denote the unique identifier of a basic block \( B_{\square} \).

4 Modeling

We define our execution model in Section 4.1. Then, in Section 4.2, we define an attacker model for test inversion and jump attacks.

4.1 Modeling Execution

We define a program execution (a program run) by a finite sequence of events, a trace. Such event-based modeling of program executions is appropriate for monitoring actual events of the program. Events mark important steps in an execution. We consider parametric events of the form \( e(p_1, \ldots, p_n) \), where \( e \) is the event name, and \( p_1, \ldots, p_n \) is the list of symbolic parameters that take some concrete values at runtime.

As we consider fault injection attacks, then events themselves are under threat (e.g., an attacker may skip an event emission). Skipping an event may result in a monitor reporting a false attack. In order to ensure events emission, we assume that the program under verification is instrumented so that every event is consecutively emitted twice.

We define the following events which have to be emitted, two consecutive times each, during a program execution, where \( G = (V, E) \) is the CFG of the program:

- For every basic block \( B \in V \), event \_begin(i) has to be emitted at the beginning of \( B \).
- For every basic block \( B \in V \), event \_end(i) has to be emitted:
  - just before instruction \_return, if the tail of \( B \) is \_return.
  - at the beginning of \( B_L \), if the tail of \( B \) is an unconditional jump instruction \_goto L. Note that, in this case, events \_end(i) have to be emitted before event \_begin(i_L).
  - at the beginning of both \( B_T \) and \( B_F \), if the tail of \( B \) is a conditional jump instruction \_ifZ z \_goto L. Note again that, in this case, events \_end(i) have to be emitted before events \_begin(i_T) and \_begin(i_F).
  - at the end of \( B \), otherwise.
- For every loop \( B_1, \ldots, B_n \) in graph \( G \), events \_reset(i_1), \ldots, \_reset(i_n) have to be emitted at the end of \( B_n \). Event \_reset(i) means that the basic block whose identifier is \( i \) may be executed again as it is involved in a loop. Note that, in this case, \_reset events have to be emitted before event \_end(i_n) as the latter is used to detect jump attacks.
- For every basic block \( B \in V \) that ends with a test \( A \), events \_bT(i, x, y) and \_bF(i, x, y) have to be emitted at the beginning of \( B_T \) and \( B_F \), respectively, where \( \text{cond}(A) = (x \ oprel y) \). We note that, in this case, the identifier \( i \) also identifies the test \( A \) and the logical operator \_oprel as a basic block can contain at most one test.

We define a program execution as follows.

Definition 1. (Program Execution). Let \( P \) be a program. An execution \( P_{\text{exec}} \) of \( P \) is a finite sequence of events \( e_1, \ldots, e_n \), where \( n \in \mathbb{N} \), such that \( e_j \in \Sigma_{\text{ALL}} = \{ \_begin(i), \_end(i), \_reset(i), \_bT(i, x, y), \_bF(i, x, y) \} \) for every \( j \in \{1, \ldots, n\} \).

For an execution \( P_{\text{exec}} \), we use the functional notation \( P_{\text{exec}}(i) \) to refer to the trace obtained from \( P_{\text{exec}} \) by considering only the related events depending on the values taken by \( i \). Indeed, \( P_{\text{exec}}(i) \) contains only the event related to the basic block identified by \( i \).
4.2 Modeling Attacker

We focus on test inversion and jump attacks. A test inversion attack is an attack where the result of a test is inverted. Whereas, a jump attack is an attack that directs the control flow of a program execution in a way that results in a path that does not exist inside the CFG of the program.

Test inversion and jump attacks can be performed using physical means [8], such as voltage and clock glitches, and electromagnetic and laser perturbations, to disturb program executions. An attack can also result from transient errors or malicious software.

We consider the multiple fault injection model for test inversion attacks, whereas we consider the single fault injection model for jump attacks. Indeed, the scenario where a jump from a basic block B into a basic block B' that is directly followed by a jump from B' into B may not be detected by our monitors. Note that the limitation of our monitors in detecting jump attacks in case of multiple fault injections is restricted to the case where the injections result in multiple jump attacks. Nevertheless, scenarios where there is only one jump attack and (possibly) other attacks, such as test inversion attack and event skip attack (provided that at most one occurrence of an event is skipped), can be detected by our monitors.

Furthermore, we assume that the attacker can skip at most one of the two consecutive occurrences of an event. Otherwise, monitors may not receive all the necessary events to output correct verdicts.

**Test inversion attack.** Consider a basic block B that ends with a test $A = z := (x \text{ oprel } y)$, $\text{if} z \text{ goto } L$. There is a test inversion attack on $A$ when $B_T$ is executed when $(x \text{ oprel } y) = \text{False}$, or when $B_F$ is executed when $(x \text{ oprel } y) = \text{True}$. In practice, the result of $A$ can be inverted, for example, by:

- skipping the conditional branch instruction, so that $B_T$ is executed regardless whether $(x \text{ oprel } y)$ evaluates to $\text{True}$ or $\text{False}$,
- skipping the instruction that involves the logical expression provided that variable $z$ already holds the value that results in branch inversion, or
- flipping the value of $z$ after the logical expression being evaluated.

**Definition 2. (Test Inversion Attack).** Let $P$ be a program, and let $P_{\text{exec}} = e_1, \ldots, e_n$ be an execution of $P$. We say that there is a test inversion attack on $P_{\text{exec}}$ if it violates $R_1$ or $R_2$, which are defined as follows, where $i$ identifies $\text{oprel}^i$:

- $R_1$: for every $j$, if $e_j = eT(i, x, y)$ then $(x \text{ oprel}^i y) = \text{True}$.
- $R_2$: for every $j$, if $e_j = eF(i, x, y)$ then $(x \text{ oprel}^i y) = \text{False}$.

**Jump attack.** In our model, a jump attack interrupts an execution of a basic block, starts an execution of a basic block not at its first instruction, or results in an edge that does not exist in the CFG. In practice, a jump attack can be performed, for example, by manipulating the target address of a branch or return. Note that we do not consider intra-basic block jumps (which are equivalent to skipping one or more instruction inside the same basic block).

Let $B$ be a basic block, and consider only events $\text{begin}(i)$ and $\text{end}(i)$. Then, in the absence of jump attacks, an execution of $B$ results in one of the following traces depending on whether none, one, or two events are skipped (assuming events duplication):

- $tr_1 = \text{begin}(i).\text{begin}(i).\text{end}(i).\text{end}(i)$,
- $tr_2 = \text{begin}(i).\text{end}(i).\text{end}(i)$,
- $tr_3 = \text{begin}(i).\text{begin}(i).\text{end}(i)$, or
- $tr_4 = \text{begin}(i).\text{end}(i)$.

During a program execution, $B$ may get executed more than once only if it is involved in a loop. In this case, between every two executions of $B$ event $\text{reset}(i)$ should be emitted.

**Definition 3. (Jump Attack).** Let $P$ be a program, and let $P_{\text{exec}}$ be an execution of $P$. Let $P_{\text{exec}}^i = e_1, \ldots, e_n$ denote the trace obtained from $P_{\text{exec}}(i)$ by filtering out all the events that are not in $\Sigma_j = \{\text{begin}(i), \text{end}(i), \text{reset}(i)\}$. We say that there is a jump attack on $P_{\text{exec}}$ if there exists $i$ such that $P_{\text{exec}}^i$ violates $R_3$, $R_4$, or $R_5$, which are defined as follows:

- $R_3$: for every $j$, if $e_j = \text{begin}(i)$ then
- $e_{j+1} = \text{end}(i)$, if $e_{j-1} = \text{begin}(i)$.
- $e_{j+1} = \text{end}(i)$, or $e_{j+1} = \text{begin}(i)$ and $e_{j+2} = \text{end}(i)$, if $e_{j-1} \neq \text{begin}(i)$.

- $R_4$: for every $j$, if $e_j = \text{end}(i)$ then
  - $e_{j-1} = \text{begin}(i)$, if $e_{j+1} = \text{end}(i)$.
  - $e_{j-1} = \text{begin}(i)$, or $e_{j-1} = \text{end}(i)$ and $e_{j-2} = \text{begin}(i)$, if $e_{j+1} \neq \text{end}(i)$.

- $R_5$: there is no $j$ such that $e_j = \text{end}(i)$ and $e_{j+1} = \text{begin}(i)$.

Definition 3 considers the jump attacks that result in executions that cannot be built by concatenating elements from \{tr1, tr2, tr3, tr4, reset(i)\}. Namely, it considers the following attacks:

- Any attack that interrupts an execution of a basic block $B$. This attack results in one or two consecutive occurrences of event $\text{begin}(i)$ that is not directly followed by event $\text{end}(i)$, which violates requirement $R_3$ of Definition 3.
- Any attack that starts the execution of a basic block $B$ not from its beginning. This attack results in event $\text{end}(i)$ that is not directly preceded by event $\text{begin}(i)$, which violates requirement $R_4$ of Definition 3.
- Any attack that performs a backward jump from the end of a basic block $B_2$ into the beginning of a basic block $B_1$ (i.e., the execution already went through $B_1$ before reaching $B_2$) such that there is no edge from $B_2$ to $B_1$ inside the related CFG. This attack results in two executions of $B_1$ that are not separated by, at least, an emission of event reset(i). Thus, it results in event $\text{begin}(i)$, which violates requirement $R_5$ of Definition 3. Note that similar forward jumps are not considered by Definition 3 as they do not violate $R_3, R_4$ nor $R_5$.

Finally, we note that a trace $P_{\text{exec}}(1')$ that starts with event reset(i) or contains more than two consecutive occurrences of event reset(i) does violate any of the requirements $R_3, R_4$ and $R_5$. However, such a trace is produced only if there is a basic block $B'$, with $i' \neq i$, that is executed not from its beginning. Consequently, $P_{\text{exec}}(1')$ violates $R_4$ in this case, and thus the attack that can result in such situation is considered by Definition 3.

Example 5 (Number of Events). In order to check verifyPIN for test inversion attacks, it has to be instrumented to produce 8 events (4 $\text{begin}(i, x, y)$ events and 4 $\text{begin}(i, x, y)$ events) since verifyPIN contains two tests and every event has to be emitted twice. Whereas, to check verifyPIN for jump attacks, it has to be instrumented to produce 24 events (10 $\text{begin}(i)$ events and 14 $\text{end}(i)$ events) since verifyPIN has 5 basic blocks, contains two conditional branches, and every event has to be emitted twice.

5 Monitors

We propose monitors that check for the presence/absence of test inversion and jump attacks on a given execution.

We assume that each event is consecutively emitted twice. Note that, in the absence of event skip attack, our monitors can still detect test inversion and jump attacks if each event is emitted only once\(^2\). Note also that our monitors can be easily modified to report event skip attack by using a variable to count the number of received events or by tracing the visited states\(^3\).

5.1 A Monitor for Detecting Test Inversions

Figure 3 depicts monitor $M_{\text{TI}}$, a QEA that detects test inversion attacks on a given execution. The alphabet of $M_{\text{TI}}$ is $\Sigma_{\text{TI}} = \{eT(i, x, y), eF(i, x, y)\}$. Monitor $M_{\text{TI}}$ has only one state, which is an accepting square state. It fails when event $eT(i, x, y)$ is emitted while $(x \text{ oprel}^1 y) = \text{False}$ (i.e., if the requirement $R_1$ is violated), or when event $eF(i, x, y)$ is emitted while $(x \text{ oprel}^1 y) = \text{True}$ (i.e., if the requirement $R_2$ is violated). $M_{\text{TI}}$ accepts multiple occurrences of events $eT(i, x, y)$ and $eF(i, x, y)$ as long as the related guards hold. Note that parameter $i$ is used to identify oprel\(^1\) and it allows reporting the test that has been inverted in case of failure.

**Proposition 1.** Let $P$ be a program and $P_{\text{exec}}$ an execution of $P$. Monitor $M_{\text{TI}}$ rejects $P_{\text{exec}}$ iff there is a test inversion attack on $P_{\text{exec}}$.

**Example 6** (Test Inversion Attack). An attacker can perform a test inversion attack on verifyPIN by skipping the second conditional branch: \texttt{if } $\text{ byteArrayCompare } \text{ returns } \text{a value } \text{BOOL_FALSE} \text{ (which

\(^2\)A smaller monitor can be used for jump attacks in this case, see Figure 4.

\(^3\)An additional state has to be added to $M_{\text{TI}}$ in this case, see Figure 3.
Figure 3: $M_{T1}$, a QEA detecting test inversion attacks.

is assigned to _t1). Thus, value 0 is assigned to variable _t2 as a result of the logical instruction _t2:=(_t1 == BOOL_TRUE). At this point, if the conditional branch is skipped, the execution branches to B3 (the success branch) instead of B4 (the failure branch) which was supposed to be executed as the value of _t2 is 0. Provided that verifyPIN is instrumented as described in Section 4.1, the faulted execution after filtering out any event that is not in $\Sigma_{IT}$ is as follows, where I$_3$ is the identifier of B3, and initially we have g_ptc=3:

$$eT(I_1, 3, 0).eT(I_1, 3, 0).eT(I_2, BOOL_FALSE, BOOL_TRUE) .eT(I_2, BOOL_FALSE, BOOL_TRUE)$$

Note that eT(I$_1$, 3, 0) is emitted at the beginning of B2, the success branch of the first test. It takes the arguments I$_1$, 3, and 0 since the corresponding test is inside B1, and the involved condition is (g_ptc > 0) where g_ptc = 3. On the other hand, eT(I$_2$, BOOL_FALSE, BOOL_TRUE) is emitted at the beginning of B3, the success branch of the second test. It takes the arguments I$_2$, BOOL_FALSE, and BOOL_TRUE since the corresponding test is inside B2, and the involved condition is (_t1 == BOOL_TRUE) where byteArrayCompare returns BOOL_FALSE into _t1 (as g_userPin is a wrong PIN).

The faulted execution is sliced by $M_{T1}$, based on the values that i can take, into the following two slices:

$$i \mapsto I_1 : eT(I_1, 3, 0).eT(I_1, 3, 0)$$
$$i \mapsto I_2 : eT(I_2, BOOL_FALSE, BOOL_TRUE).eT(I_2, BOOL_FALSE, BOOL_TRUE)$$

Slice i $\rightarrow$ I$_1$ satisfies both requirements R$_1$ and B$_2$, and thus it is accepted by $M_{T1}$($I_1$). While, slice i $\rightarrow$ I$_2$ does not satisfy the requirement R$_1$ as event eT(I$_2$, BOOL_FALSE, BOOL_TRUE) is emitted but (BOOL_FALSE == BOOL_TRUE) = False, and thus it is rejected by $M_{T1}$($I_2$). Indeed, the occurrence of eT(I$_2$, BOOL_FALSE, BOOL_TRUE) leads into an implicit failure state since the related guard is not satisfied. Therefore, since slice i $\rightarrow$ I$_2$ is rejected by $M_{T1}$($I_2$), the whole faulted execution is rejected by $M_{T1}$.

5.2 A Monitor for Detecting Jump Attacks

Figure 4 depicts monitor $M_{J}$, a QEA that detects jump attacks on a given program execution. The alphabet of $M_{J}$($i$) is $\Sigma_{J}$ = {begin($i$), end($i$), reset($i$)}. Monitor $M_{J}$ covers every basic block $i$ inside the CFG of the given program. An instantiation of i results in the EA $M_{J}$($i$). Note that $M_{J}$ assumes a single fault injection model.

Figure 4: $M_{J}$, a QEA detecting jump attacks.

**Proposition 2.** Let P be a program, and let $P_{exec}$ be an execution of P. Monitor $M_{J}$ rejects $P_{exec}$ iff there is a jump attack on $P_{exec}$.

Indeed, $M_{J}$ cannot output a final verdict concerning R$_1$ until the end of the execution as event end($i$) may occur at any point in the future. One way to explicitly catch the end of an execution is to include event exit in $\Sigma_{J}$, and add a transition from state (5), labeled by exit, to a new accepting square state, say state (6). Then, an occurrence of event exit in state (5) means that event end($i$) will definitely not occur, and thus leads to an implicit failure state. A self-loop on state (4) labeled by exit is also required. Note that an occurrence of event exit at state (1) will also lead to failure.
**Example 7 (Jump Attack).** An attacker can perform a jump attack on `verifyPIN` by modifying the return address of the function `byteArrayCompare`, for example, into the address of the first instruction (`g_ptc := 3`) of the basic block `B_3` (see Figure 2). This attack interrupts the execution of `B_2`, and allows the attacker to get authenticated with a wrong PIN as it skips the test that follows `byteArrayCompare`. Consequently, `B_3` (the success branch) will be executed regardless of the value returned by `byteArrayCompare`. Provided that `verifyPIN` is instrumented as described in Section 4.1, the faulted execution after filtering out any event that is not in `Σ_2` is as follows, where `I_j` is the identifier of `B_j`:

\[
\begin{align*}
&begin(I_1).begin(I_1).end(I_1).begin(I_2).begin(I_2).begin(I_3) \\
&.begin(I_3).end(I_3).end(I_3).begin(I_4).begin(I_5).begin(I_6).end(I_6)
\end{align*}
\]

The faulted execution is sliced by \(M_3\), based on the values that \(i\) can take, into the following four slices:

\[
\begin{align*}
&i \mapsto I_1: begin(I_1).begin(I_1).end(I_1), \\
&i \mapsto I_2: begin(I_2).begin(I_2), \\
&i \mapsto I_3: begin(I_3).begin(I_3).end(I_3).end(I_3), \text{ and} \\
&i \mapsto I_5: begin(I_5).begin(I_5).end(I_5).end(I_5).
\end{align*}
\]

Slices \(i \mapsto I_1, i \mapsto I_2, \) and \(i \mapsto I_5\) satisfy requirements \(R_5, R_4\) and \(R_6\). Thus, they are respectively accepted by \(M_3(I_1), M_3(I_3), \) and \(M_3(I_5)\). However, slice \(i \mapsto I_2\) does not satisfy the requirement \(R_5\) as it contains two consecutive occurrences of event `begin(I_2)` that are not followed by event `end(I_2)`. Thus it is rejected by \(M_3(I_2)\). Indeed, the first occurrence of event `begin(I_2)` fires the transition from state (1) into state (2), and the second occurrence of `begin(I_2)` fires the transition from state (2) into state (3), see Figure 4. Thus, \(M_3(I_2)\) ends in state (3), which is a failure state. Therefore, since slice \(i \mapsto I_2\) is rejected by \(M_3(I_2)\), the whole faulted execution is rejected by \(M_3\).

Note, given a CFG \((V, E)\), monitor \(M_3\) cannot detect an attack where a forward jump from the end of `B \in V` into the beginning of `B' \in V` with \((B, B') \notin E\) is executed. Indeed, in order to detect such a jump, a global monitor with a structure similar to the CFG is needed where basic blocks are replaced by EAs that resemble \(M_3(1)\) with the adjustment of reset transitions in accordance with the loops.

### 6 Monitor Validation

We validate our monitors by demonstrating their effectiveness in detecting simulated attacks against \(P = \) `verifyPIN`. Following the initial C implementation, we have implemented `verifyPIN` and the monitors using Java and AspectJ\(^4\). We have instrumented `verifyPIN` at the source code level. More precisely, for every required event we have defined an associated function which is called inside `verifyPIN` at the positions where the event has to be emitted as specified in Section 4.1. The associated functions are used to define pointcuts in AspectJ. When a function is called, i.e., a pointcut is triggered, the corresponding event is fed to the running monitor. The monitor then makes a transition based on its current state and the received event, and reports a verdict. The code segments executed by the monitor (called advices) are woven within the original source files to generate the final source code that is compiled into an executable.

For example, Listing 3 depicts the Java implementation of \(M_7\). Two states are defined: `Ok` (accepting state) and `Error` (failure state). Function `updateState` (Lines 4-19) takes an event and then, after evaluating the condition \((x \operatorname{op} \_\operatorname{rel} y)\), it updates the `currentState`. If the monitor is in state `Ok`, the state is updated into `Error` if the received event is `eF` and the condition evaluates to false, or the received event is `eF` and the condition evaluates to true. Once state `Error` is reached, the monitor cannot exit from it. Function `currentVerdict` (Lines 21-26) emits verdict `CURRENTLY_TRUE` if the `currentState` is `Ok`, whereas it emits verdict `FALSE` if the `currentState` is `Error`.

In what follows, we illustrate about how we carried out our experiment and the obtained results. The experiment was conducted using Eclipse 4.11 and Java JDK 8u181 on a standard PC (Intel Core i7 2.2 GHz, 16 GB RAM).

#### 6.1 Normal Executions

Providing events duplication, running instrumented `verifyPIN` in the absence of an attacker results in one of the following 3 executions depending on the values of `g_ptc` and `g_userPin`:

- \(P_{\text{exec}}\), which contains 10 events: 4 events `begin`, 4 events `end` and 2 events `eF` that result from executing `B_1` and `B_5` (see Figure 2). This execution is performed when `g_ptc \leq 0`.

\(^4\) [www.eclipse.org/aspectj/](http://www.eclipse.org/aspectj/)
Listing 3: Java implementation of $M_T$.

```java
public class VerificationMonitorTI {
    private State currentState = State.Ok;

    public void updateState(Event e) {
        switch (this.currentState) {
            case Ok:
                int x = e.getX();
                int y = e.getY();
                String oprel = e.getOprel();
                boolean condition = evaluateCond(x, y, oprel);
                if ((e.getName().equals("eT") && !condition) || (e.getName().equals("eF") && condition))
                    { this.currentState = State.Error; }
                break;
            case Error:
                // No need to execute any code.
                break;
        }
        System.out.println("moved to " + this.currentState);
    }

    public Verdict currentVerdict () {
        switch (this.currentState) {
            case Ok: return Verdict.CURRENTLY_TRUE;
            case Error: return Verdict.FALSE;
            default: return Verdict.FALSE;
        }
    }

    // ...
```

The time overhead depends on the size of the application and the number of events. The size of `verifyPIN` is small; hence the measured overhead represents an extreme and unfavorable situation. A more representative measure of the overhead should be done on larger applications, especially because our approach aims at protecting the critical parts of an application instead of protecting every single instruction. Moreover, we note that our implementation is not optimized yet, and that using AspectJ for instrumentation is not the best choice performance-wise. Instrumentation causes most of the overhead (see Table 1). In the future, using tools such as ASM [22] to directly instrument the bytecode would result in a smaller overhead.

Table 1 summarizes the cumulative execution time and the memory footprint of 100K runs of $P_{exec}$: (i) without instrumentation, (ii) with instrumentation, and (iii) with instrumentation and the monitors. Note that the memory consumption can be bounded as a monitor processes only one event at a time, keeps track only of the current state, and are parametrized by the current executing block.\footnote{We verified empirically that the memory consumption is insensitive to the number of events. However, we did not report the numbers because of lack of space.}

![Table 1: Cumulative execution time and memory footprint of 100K runs of $P_{exec}$.

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (ms)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>verifyPIN</code></td>
<td>164</td>
<td>284</td>
</tr>
<tr>
<td><code>Inst. verifyPIN</code></td>
<td>209 ($\times$ 1.27)</td>
<td>796.6 ($\times$ 2.8)</td>
</tr>
<tr>
<td><code>Inst. verifyPIN &amp; Monitors</code></td>
<td>228 ($\times$ 1.39)</td>
<td>801 ($\times$ 2.82)</td>
</tr>
</tbody>
</table>
6.2 Test Inversion Attack

Function verifyPIN contains two tests. The first test: 
_\text{t}_0 := (g\_ptc > 0), ifZ \_t0 goto L1 is represented, in Java bytecode, using the instruction ifle L1. The instruction ifle L1 compares g\_ptc and 0, which are previously loaded into the stack, and performs a branch into L1 if g\_ptc is less than or equal to 0. This test can be inverted by replacing ifle L1 with ifgt L1, which performs a branch if g\_ptc is greater than 0.

Similarly, the second one: 
_\text{t}_2 := (_t1 == BOOL_TRUE), ifZ \_t2 goto L2 is represented using the instruction if_icmpne L2, which compares _t1 and BOOL_TRUE, and performs a branch into L2 if they are not equal. This test can be inverted by replacing if_icmpne with if_icmpne, which performs a branch if the operands are equal.

The binary opcodes of ifle, ifgt, if_icmpne and if_icmpne are respectively “1001 1110”, “1001 1101”, “1010 0000” and “1001 1111”. Hence, replacing ifle with ifgt (resp. if_icmpne with if_icmpne) requires modifying 2 bits (resp. 6 bits).

Depending on the values of g\_ptc and g\_userPin, a test inversion attack can be used, e.g., to force authentication with a wrong PIN or to prevent the authentication with the correct PIN.

As verifyPIN contains two tests, we consider the two following scenarios:

- In case of a wrong user PIN in the first trial (i.e., g\_ptc = 3), inverting the second test results in a successful authentication. This is the attack presented in Example 6. Note that monitor \_M\_TI reports an attack after processing the first occurrence of event eT corresponding to the second test as the related guard does not hold in this case. That is after receiving 11 events: 4 events begin, 4 events end and 3 events eT. Whereas, the full execution contains 20 events.

- In case of a wrong user PIN in the fourth trial (i.e., g\_ptc = 0), inverting both tests results in a successful authentication. Again, \_M\_TI reports an attack after processing the first occurrence of event eT. That is after receiving 5 events: 2 events begin, 2 events end and 1 events eT.

Forcing the “success branch”, in the scenarios above, can be also performed by replacing ifle and/or if_icmpne with nop, which is equivalent to instruction skip, and thus results in the execution of the the “success branch” regardless of the operands’ values. Replacing ifle (resp. if_icmpne) with nop (“0000 0000”, in binary) requires modifying 5 bits (resp. 2 bits). Note that replacing ifle or if_icmpne with nop only works with Java 6 or earlier\(^6\).

Our experiment showed that \_M\_TI can detect both attack scenarios presented above.

6.3 Jump Attack

A jump attack can be simulated, in Java bytecode, by replacing an instruction with goto L for a certain line number L. However, this results in an inconsistent stack map frame for Java 7 and latest versions. Nevertheless, it is possible to simulate the jump attack presented in Example 7 at the source code level \(^7\). We note here that the main purpose is not performing the attack, but to validate that \_M\_J can detect jump attacks. The latter is confirmed by our experiment.

The faulted execution presented in Example 7 contains 16 events: 6 events begin, 6 events end and 2 events eT. However, \_M\_J reports an attack after processing the first occurrence of event end corresponding to B3. That is, after receiving 9 events: 4 events begin, 3 events end and 2 events eT. Note that the execution of B3 has been interrupted before, but \_M\_J cannot report an attack in this case until the end of the execution as event end may appear at any time in the future.

7 Related Work

This work introduces formal runtime verification monitors to detect fault attacks. Runtime verification/monitoring was successfully applied to several domains, e.g., for monitoring financial transactions \(^24\), monitoring IT logs \(^25\), monitoring electronic exams \(^26\), monitoring smart homes \(^27\).

\(^6\) Starting from Java 7, the typing system requires a stack map frame at the beginning of each basic block \(^23\). Thus, a stack map frame is required by every branching instruction. The stack map frame specifies the verification type of each operand stack entry and of each local variable. Replacing if.icmpne with if.icmpne does not result in a violation of the related stack map frame, however, replacing it with nop does. Hence, in order to simulate the attack by replacing if_icmpne with nop, the stack map frame also has to be modified.

\(^7\) Indeed, it is not possible to simulate this attack by modifying the return address of byte\_Array\_Compare. However, one can simulate the effect of goto using break and continue statements.
In the following, we compare our work to the research endeavors that propose software-based protections against attacks. We distinguish between algorithm-level and instruction-level approaches.

### 7.1 Algorithm-level Approaches

At the algorithm level, there are approaches that use basic temporal redundancy [28, 1, 11, 8, 29] such as computing a cryptographic operation twice then comparing the outputs. There are also approaches that use parity codes [10] and digest values [30].

Some other works make use of signature mechanisms and techniques to monitor executions, such as state automata and watchdog processor, in order to detect errors or protect the executions control flow. For instance, [31, 32, 33] use watchdogs for error detection at runtime. The underlying principle is to provide a watchdog processor [34] with some information about the process (or processor) to be verified. Then, the watchdog concurrently collects the relevant information at runtime. An error is reported when the comparison test between the collected and provided information fails. The watchdog processor can also be used to detect control flow errors, as illustrated in [32], by verifying that the inserted assertions are executed in the order specified by the CFG. Ersoz et al. [31] propose the watchdog task, a software abstraction of the watchdog processor, to detect execution errors using some assertions. Saxena et al. [33] propose a control-flow checking method using checksums and watchdog. To ensure the control flow of a program, a signature is derived from the instructions based on checksum. However, these approaches focus on protecting solely basic blocks and the control flow, and do not protect branch instructions. Whereas our monitors detect attacks on branch instructions. Moreover, using a watchdog processor for monitoring involves larger communication overhead than using runtime verification.

Nicolescu et al. [35] propose a technique (SIED) to detect transient errors such as bit-flip. This technique has been designed to be combined with an instruction duplication approach. It performs comparison checks and uses signatures to protect the intra-block and inter-block control flow, respectively. Relaying on comparison checks make it subject to fault injections that skip the check itself [36, 37]. Note that experiments have shown that SIED cannot detect all bit-flip faults. Later in [38], the authors propose another error-detection mechanism that provides full coverage against single bit-flip faults. These works only consider single bit-flip as a fault model, whereas our definition of the attacks is independent of the technique used to perform the attack.

Sere et al. [39] propose a set of countermeasures based on basic block signatures and security checks. The framework allows developers to detect mutant applications given a fault model, and thus developing secure applications. The countermeasures can be activated by the developer using an annotation mechanism. This approach requires some modifications in the Java Virtual Machine in order to perform the security checks. Bouffard et al. [40] propose an automaton-based countermeasure against fault injection attacks. For a given program, every state of the corresponding automaton corresponds to a basic block of the CFG, and each transition corresponds to an edge, i.e., allowed control flow. Thus the size of the resulting automaton is proportional to the size of the CFG of the program. Whereas, our monitors are lightweight and small in size.

Fontaine et al. [41] propose a model to protect control flow integrity (CFI). The approach relies on instrumenting the LLVM IR code, and then using an external monitor (state automaton) which enforces CFI at runtime. Lalande et al. [42] propose an approach to detect intra-procedural jump attacks at source code level, and to automatically inject some countermeasures. However, these approaches do not consider test inversion attacks.

Algorithm-level approaches do not require changes to the instruction code. However, they are not effective against compile-time modifications. Moreover, it has been shown that they are not robust against multiple fault injections [11, 12] or skipping the critical parts of the code [36, 37]. Furthermore, most of the existing algorithm-level approaches are not generic.

Our approach is based on the formal model of QEAs, which is one of the most expressive and efficient form of runtime monitor [43]. Our approach is also generic as it can be applied to any application and the monitors can be easily tweaked and used in combination with the monitors for the program requirements. Moreover, monitors may run in a hardware-protected memory as they are lightweight and small in size. This provides more protection against synchronized multiple fault injections on both the monitored program and the monitor. Furthermore, runtime monitoring is modular and compatible with the existing approaches, for instance, monitor $\mathcal{M}_{TSE}$ can be used to detect fault attacks on the test that compares the outputs when an operation is computed twice.

Note, to ensure the correct extraction of the necessary information (i.e., events) from a running program, we use emission duplication. This may require the duplication of every related instruction as duplication at the source code level may not be sufficient.
### 7.2 Instruction-level Approaches

At instruction-level, there are approaches that aim at providing fault-tolerance. These include (i) the approaches that apply the duplication or triplication of instructions [13, 14] in order to provide 100% protection against skip fault injections, and (ii) the approaches that rely on replacing every instruction with a sequence of functionally equivalent instructions such that skipping any of these instructions does not affect the outcome [15, 44]. Such approaches provide more guarantees than algorithm-level approaches. Indeed, it is believed that they are robust against multiple fault injections under the assumption that skipping two consecutive instructions is too expensive and requires high synchronization capabilities [13, 15]. However, they require dedicated compilers for code generation. Moreover, approaches that apply the duplication or replacement of instructions are processor dependent as some instructions have to be replaced by a sequence of idempotent or functionally equivalent instructions. Furthermore, they introduce a large overhead in performance and footprint. For instance, instruction duplication increases the overhead at least twice. Nevertheless, the overhead can be decreased by protecting only the critical parts of the code. In comparison, our monitors are easy to implement and deploy, introduce smaller overhead, and are independent of the processor and the compiler. Note that, runtime monitoring does not provide 100% protection against all fault attacks. Our monitors can detect test inversion and jump attacks. Providing more guarantees requires more monitors.

There are also approaches that aim at ensuring CFI. Most existing CFI approaches follow the seminal work by Abadi et al. [45], which makes use of a special set of instructions in order to check the source and destination addresses involved in indirect jumps and indirect function calls. CFI approaches do not aim to provide a 100% fault coverage. Instead they aim at providing protections against jump-oriented attacks [46, 47], and return-oriented attacks [48]. A related technique called control flow locking (CFL) has been introduced by Bletsch et al. [49] in order to provide protection against code-reuse attacks. Instead of inserting checks at control flow transfers, CFL locks the corresponding memory address before a control flow transfer, and then unlocks it after the transfer is executed.

Similar to other instruction-level approaches, these CFI approaches requires change to the instruction code, and usually introduce large overhead. Note that detecting jump attacks by our monitors is some sort of reporting CFI violations. Note also that CFI does not deal with test inversion attacks as taking any of the branches after a conditional branch does not violate CFI.

### 8 Conclusions and Perspectives

We formally define test inversion and jump attacks. Then, we propose monitors expressed as Quantified Event Automata in order to detect these attacks. Our monitors are lightweight and small in size, and they support the duplication of events emission which provides protection against event skip attacks. Finally, we demonstrate the validity of our monitors using attack examples on verifyPIN.

In the future, we will define more monitors to detect additional attacks following the principles exposed in this paper. For example, a monitor that can detect attacks on function calls. We also plan (i) to verify applications larger than verifyPIN, combined with detailed feasibility and performance analysis, (ii) to use a Java bytecode editing tool, such as ASM [22] or JNIF [50], to simulate faults, and (iii) to deploy the monitors on hardware architectures such as smart cards, Raspberry Pi, and microcontrollers based on Arm Cortex-M processor. Furthermore, we consider building a tool for automatic generation of monitors from QEAs, and developing a runtime enforcement [51, 52, 53] framework where some corrective actions and countermeasures are automatically executed and taken respectively once an attack is detected.

### References


A Proofs

Proposition 1. Let \( P \) be a program and \( P_{exec} \) an execution of \( P \). Monitor \( M_{TI} \) rejects \( P_{exec} \) iff there is a test inversion attack on \( P_{exec} \).

Proof. Assume that \( M_{TI} \) rejects \( P_{exec} \). We have to show that there is a test inversion attack on \( P_{exec} \), i.e., \( P_{exec} \) violates \( R_1 \) or \( R_2 \). As \( M_{TI} \) rejects \( P_{exec} \), then there exists \( i \) such that \( M_{TI}(i) \) fails (i.e., ends in a failure state). Thus, \( M_{TI}(i) \) fires a transition into an implicit failure state since \( M_{TI}(i) \) has only one (explicit) state which is an accepting state. This means that \( P_{exec} \) contains, for some \( x \) and \( y \), event \( eT(i, x, y) \) such that \( (x \; \text{oprel} \; 1 \; y) = \text{False} \) which violates \( R_1 \), or event \( eF(i, x, y) \) such that \( (x \; \text{oprel} \; 1 \; y) = \text{True} \) which violates \( R_2 \). So, \( P_{exec} \) violates \( R_1 \) or \( R_2 \), and thus there is a test inversion attack on \( P_{exec} \). Hence, we can conclude for the second direction.

To prove the second direction, we assume that there is a test inversion attack on \( P_{exec} \), and we show that \( M_{TI} \) rejects \( P_{exec} \). If there is a test inversion attack on \( P_{exec} \), then \( P_{exec} \) violates \( R_1 \) or \( R_2 \). If \( P_{exec} \) violates \( R_1 \), then it contains an event \( eT(i, x, y) \) such that \( (x \; \text{oprel} \; 1 \; y) = \text{False} \), which fires a transition into an implicit failure state as the guard related to event \( eT(i, x, y) \) is not satisfied. Thus, \( M_{TI} \) fails and rejects \( P_{exec} \). If \( P_{exec} \) violates \( R_2 \) then it contains an event \( eF(i, x, y) \) such that \( (x \; \text{oprel} \; 1 \; y) = \text{True} \), which fires a transition into an implicit failure state as the guard related to event \( eF(i, x, y) \) is not satisfied. Thus, \( M_{TI} \) fails and rejects \( P_{exec} \). Hence, we conclude for the second direction and the proof is done.

Proposition 2. Let \( P \) be a program, and let \( P_{exec} \) be an execution of \( P \). Monitor \( M_{J} \) rejects \( P_{exec} \) iff there is a jump attack on \( P_{exec} \).

Proof. Assume that \( M_{J} \) rejects \( P_{exec} \). We have to show that there is a jump attack on \( P_{exec} \), i.e., there exists \( i \) such that \( P_{exec}^i = e_1, \ldots, e_n \) violates \( R_3 \) or \( R_4 \). As \( M_{J} \) rejects \( P_{exec} \), then there exists \( i \) such that \( E_A M_{J}(i) \) fails while consuming \( P_{exec}^i \). We split the cases in which \( H_{J}(i) \) fails.

- \( H_{J}(i) \) has encountered event \( end(i) \) in state (1). This means that there exists \( j \) such that \( e_j = \text{end}(i) \), and that \( e_{j-1} \) is either \( \epsilon \) or \( \text{reset}(i) \) since state (1) is the initial state and it can only be reached by \( \text{reset}(i) \). Thus, \( P_{exec}^i \) violates \( R_4 \).

- \( H_{J}(i) \) ends in state (2). Then, in this case, there exists \( j \) such that \( e_j = \text{begin}(i) \) since state (2) can only be reached from state (1) through event \( \text{begin}(i) \). Moreover, we can deduce that \( e_{j+1} \) is neither \( \text{begin}(i) \) nor \( \text{end}(i) \). Thus, \( P_{exec}^i \) violates \( R_3 \).

- \( H_{J}(i) \) has encountered event \( e_j = \text{reset}(i) \) in state (2). State (2) can only be reached from state (1) through event \( \text{begin}(i) \). Then, \( e_{j-1} = \text{begin}(i) \). So, there exists \( e_{j-1} = \text{begin}(i) \) such that \( e_j \) is neither \( \text{end}(i) \) nor \( \text{begin}(i) \) which violates \( R_3 \).
Hence, there exists \( i \) such that \( P_{\text{exec}}(i) \) violates \( R_3 \), \( R_4 \) or \( R_5 \), and we can conclude about the first direction.

To prove the second direction, we assume that there exist \( i \) such that \( P_{\text{exec}}(i) \) violates \( R_3 \), \( R_4 \) or \( R_5 \) and we show that \( M_i \) rejects \( P_{\text{exec}} \). We split cases according which requirement is violated.

- If \( P_{\text{exec}}(i) \) violates \( R_3 \), then it contains an event \( e_j = \text{begin}(i) \) for some integer \( j \) such that:
  
  - \( e_{j-1} = \text{begin}(i) \) and \( e_{j+1} \neq \text{end}(i) \). In this case if, before receiving \( e_{j-1}, M_i(\text{begin}(i)) \) was:
    
    * in state (1). Then \( e_{j-1} = \text{begin}(i) \) leads into state (2) and \( e_j = \text{begin}(i) \) leads into state (3). As \( e_{j+1} \neq \text{end}(i) \) and from state (3) there is only one explicit transition labeled by \( \text{end}(i) \), then \( M_i(\text{end}(i)) \) ends in state (3) (i.e., \( e_{j+1} = \epsilon \)) or a transition into an implicit failure state is fired.
    
    * in state (2). Then \( e_{j-1} = \text{begin}(i) \) leads into state (3), and \( e_j = \text{begin}(i) \) leads into an implicit failure state.
    
    * in state (3), (4) or (5). Then \( M_i(\text{end}(i)) \) fails since none of the states (3), (4) and (5) has an explicit outgoing transition labeled by \( \text{begin}(i) \).
  
  - or \( e_{j-1} \neq \text{begin}(i) \) and \( e_{j+1} \neq \text{end}(i) \), and \( e_{j+2} \neq \text{end}(i) \). This is equivalent to \( e_{j-1} \neq \text{begin}(i) \) and, \( e_{j+1} = \epsilon \) (i.e., \( j = 1 \)) or \( e_{j+1} = \text{reset}(i) \) or \( "e_{j+1} = \text{begin}(i) \) and \( e_{j+2} \neq \text{end}(i) \" since \( \Sigma_j = \{ \text{begin}(i), \text{end}(i), \text{reset}(i) \} \). In this case if, before receiving \( e_{j-1}, M_i(\text{begin}(i)) \) was:
    
    * in state (1). If \( e_{j-1} = \text{end}(i) \), a transition into an implicit failure state is fired. Otherwise, we have that \( e_{j-1} = \text{reset}(i) \) or \( \epsilon \) (i.e., \( j = 1 \)), and thus \( M_i(\text{end}(i)) \) stays in state (1). Then, \( e_j = \text{begin}(i) \) leads into state (2). Then in state (2), if \( e_{j+1} = \text{reset}(i) \) a transition into an implicit failure state is fired; if \( e_{j+1} = \epsilon \) then \( M_i(\text{end}(i)) \) ends in state (2); if \( e_{j+1} = \text{begin}(i) \) the transition from state (2) into state (3) is fired and, as \( e_{j+2} \neq \text{end}(i) \) in this case, then a transition into an implicit failure state is fired or \( M_i(\text{end}(i)) \) ends in failure state (3).
    
    * in state (2). If \( e_{j-1} = \text{reset}(i) \), a transition into an implicit failure state is fired. Otherwise, we have that \( e_{j-1} = \text{end}(i) \) which leads from state (2) into state (4). Then in state (4), as \( e_j = \text{begin}(i) \), a transition into an implicit failure state is fired.
    
    * in state (3). Similar to the case of state (2).
    
    * in state (4). If \( e_{j-1} = \text{end}(i) \), the transition into state (5) is fired. Then in state (5), as \( e_j = \text{begin}(i) \), a transition into an implicit failure state is fired. Otherwise, we have that \( e_{j-1} = \text{reset}(i) \), and thus the transition from state (4) into state (1) is fired. Then in state (1), as \( e_j = \text{begin}(i) \), the transition into state (2) is fired. Then in state (2), if \( e_{j+1} = \text{reset}(i) \) a transition into an implicit failure state is fired; if \( e_{j+1} = \epsilon \) then \( M_i(\text{end}(i)) \) ends in state (2) which is a failure state; if \( e_{j+1} = \text{begin}(i) \) the transition from state (2) into state (3) is fired and, as \( e_{j+2} \neq \text{end}(i) \) in this case, then a transition into an implicit failure state is fired or \( M_i(\text{end}(i)) \) ends in state (3), which is a failure state.
    
    * in state (5). If \( e_{j-1} = \text{reset}(i) \) the reasoning is similar to the case of state (4) when \( e_{j-1} = \text{reset}(i) \). Otherwise, we have that \( e_{j-1} = \text{end}(i) \) which leads into an implicit failure state.
Hence, If $P_{\text{exec}}^j(1)$ violates $R_3$ then $M_j(1)$ fails, and thus $M_j$ rejects $P_{\text{exec}}$.

- If $P_{\text{exec}}^j(1)$ violates $R_4$, then it contains an event $e_j = \text{end}(i)$ for some integer $j$ such that
  - $e_{j-1} \neq \text{begin}(i)$ and $e_{j+1} = \text{end}(i)$. In this case if, before receiving $e_{j-1}$, $M_j(1)$ was:
    * in state (1). If $e_{j-1}$ is $\text{reset}(i)$ the self-loop transition over state (1) is fired. Then, as $e_j = \text{end}(i)$, a transition into an implicit failure state is fired. Otherwise, if $e_{j-1}$ is $\text{end}(i)$, then a transition into an implicit failure state is fired. Otherwise, we have that $e_{j-1} = \epsilon$ (i.e., $j = 1$), and thus $e_j = \text{end}(i)$ leads from state (1) into an implicit failure state.
    * in state (2). If $e_{j-1}$ is $\text{reset}(i)$ a transition into an implicit failure state is fired. Otherwise, we have that $e_{j-1} = \text{end}(i)$, and thus the transition from state (2) into state (4) is fired. Then in state (4), event $e_j = \text{end}(i)$ leads into state (5). Then in state (5), event $e_{j+1} = \text{end}(i)$ leads into an implicit failure state.
    * in state (3). Similar to the case of state (2).
    * in state (4). If $e_{j-1} = \text{reset}(i)$ the transition to state (1) is fired. Then in state (1), $e_j = \text{end}(i)$ leads into an implicit failure state. Otherwise, we have that $e_{j-1} = \text{end}(i)$ which fires the transition from state (4) into state (5). Then in state (5), $e_j = \text{end}(i)$ leads into an implicit failure state.
    * in state (5). If $e_{j-1} = \text{reset}(i)$ the transition to state (1) is fired. Then $e_j = \text{end}(i)$ leads into an implicit failure state. Otherwise, we have $e_{j-1} = \text{end}(i)$ which fires a transition into an implicit failure state.
  - or $e_{j+1} \neq \text{end}(i)$ and $e_{j-1} \neq \text{begin}(i)$, and "$e_{j-1} \neq \text{end}(i)$ or $e_{j-2} \neq \text{begin}(i)$". This is equivalent to $e_{j+1} \neq \text{end}(i)$, and $e_{j-1} = \text{reset}(i)$ or "$e_{j-1} = \text{end}(i)$ and $e_{j-2} \neq \text{begin}(i)$" since $\Sigma_j = \{\text{begin}(i), \text{end}(i), \text{reset}(i)\}$. In this case if, before receiving $e_{j-1}$, $M_j(1)$ was:
    * in state (1). If $e_{j-1}$ is $\text{end}(i)$, a transition into an implicit failure state is fired. Otherwise, we have that $e_{j-1} = \text{reset}(i)$ or $\epsilon$ (i.e., $j = 1$), and thus $M_j(1)$ stays in state (1). Then, as $e_j = \text{end}(i)$, a transition into an implicit failure state is fired.
    * in state (2). In this case $e_{j-1}$ must be equal to $\text{reset}(i)$ since if $e_{j-1} = \text{end}(i)$, then $e_{j-2} \neq \text{begin}(i)$ and thus state (2) cannot be reached. Indeed, state (2) can only be reached from state (1) by $\text{begin}(i)$. In state (2), $e_{j-1} = \text{reset}(i)$ leads into an implicit failure state.
    * in state (3). Similar to the case of state (2).
    * in state (4). If $e_{j-1} = \text{reset}(i)$ the transition into state (1) is fired. Then in state (1), as $e_j = \text{end}(i)$, a transition into an implicit failure state is fired. Otherwise, we have that $e_{j-1} = \text{end}(i)$ which leads into state (5). Then in state (5), $e_j = \text{end}(i)$ leads into an implicit failure state.
    * in state (5). If $e_{j-1} = \text{reset}(i)$ the transition into state (1) is fired. Then, as $e_j = \text{end}(i)$, a transition into an implicit failure state is fired. Otherwise, we have that $e_{j-1} = \text{end}(i)$ which leads into an implicit failure state.

Hence, If $P_{\text{exec}}^j(1)$ violates $R_4$ then $M_j(1)$ fails, and thus $M_j$ rejects $P_{\text{exec}}$.

- If $P_{\text{exec}}^j(1)$ violates $R_5$, then it contains an event $e_j = \text{end}(i)$ for some integer $j$ such that $e_{j+1} = \text{begin}(i)$. In this case if, before receiving $e_j$, $M_j(1)$ was:
  - in state (1). Then event $e_j = \text{end}(i)$ fires a transition into an implicit failure state.
  - in state (2). Then event $e_j = \text{end}(i)$ fires the transition into state (4). In state (4), $e_{j+1} = \text{begin}(i)$ fires a transition into an implicit failure state.
  - in state (3). Similar to the case of state (2).
  - in state (4). Then $e_j = \text{end}(i)$ fires the transition into state (5). In state (5), $e_{j+1} = \text{begin}(i)$ fires a transition into an implicit failure state.
  - in state (5). Then $e_j = \text{end}(i)$ fires a transition into an implicit failure state.

Hence, if $P_{\text{exec}}^j(1)$ violates $R_5$ then $M_j(1)$ fails, and thus $M_j$ rejects $P_{\text{exec}}$.

Therefore, if there exists $i$ such that $P_{\text{exec}}^j(1)$ violates $R_3$, $R_4$ or $R_5$ then $M_j$ rejects $P_{\text{exec}}$. Thus, we can conclude for the second direction and the proof is done.