Detecting Fault Injection Attacks with Runtime Verification

Ali Kassem
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble, France
ali.kassem@inria.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble, France
ylies.falcone@inria.fr

ABSTRACT

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

ACM Reference Format:
Ali Kassem and Yliès Falcone. 2019. Detecting Fault Injection Attacks with Runtime Verification. In *3rd Software Protection Workshop (SPRO'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338503.3357724

1 INTRODUCTION

Fault injections are effective techniques to exploit vulnerabilities in embedded applications and implementations of cryptographic primitives [4, 13, 15, 20, 37, 53]. Fault injections can be thwarted (or detected) using software or hardware countermeasures [5, 33]. Hardware countermeasures are expensive, and they are unpractical for off-the-shelf products. Henceforth, software countermeasures are commonly adopted. Software countermeasures can be categorized into algorithm-level and instruction-level countermeasures.

On the one hand, algorithm-level countermeasures do not require changes to the low-level code, compiler and instructions set. They are usually rely on redundancy such as computing a cryptographic operation twice then comparing the outputs [16], or using checksums and parity bits [34]. This makes them weak against multiple fault injections and skipping critical instructions [3, 23].

SPRO'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6835-3/19/11…$15.00
https://doi.org/10.1145/3338503.3357724

On the other hand, instruction-level countermeasures require changes to the low-level code, for example by applying instruction duplication or triplication [6, 8]. Instruction-level countermeasures are more robust. Indeed, instruction duplication countermeasures are secure against multiple skip attacks under the assumption that skipping two consecutive instructions is too expensive and requires high synchronization capabilities [6, 44]. However, instruction-level countermeasures may introduce a large overhead, for instance, instruction duplication increases the overhead by at least two times [6]. Moreover, they require changing the instructions 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 [10, 26, 31, 39] and monitors to detect fault injections that result in test inversion attacks and arbitrary jumps in the control flow. We use Quantified Event Automata (QEAs) [7] to express monitors. QEAs are expressive formalism to represent parametric specifications to be checked at runtime. The proposed monitors support event duplication, which provides more protection against event skip attacks. 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 [21].

Our monitors are lightweight and small in size. The monitors execution time is proportional to the size of the monitored program. However, the memory overhead can be bounded as only data related to the "active" basic block need to be kept in memory. This makes our monitors suitable for moderate- and small-memory devices, such as smart cards and IoT devices (*e.g.,* wearables and sensors). Furthermore, a monitor may fit in a small hardware-protected memory where the entire program cannot fit. Thus, when possible, providing more protection against synchronized multiple fault injections on both the program and the monitor.

A monitor can be coupled with a program in a synchronous or an asynchronous way. In the case of synchronous monitoring, the program is instrumented in a way that permits the monitor to employ a synchronization mechanism. This may result in some delay in the program execution until the necessary checks are made by the monitor. Whereas in the case of asynchronous monitoring, the monitor still executes at the same time with the program. However, the monitor does not have any control over the program, which may result in late faults detection. In the latter, whenever a certain action is performed by the running program, a corresponding event is emitted. The monitor then receives and process the event independently from the program which continue its execution. Note

that this differs from off-line monitoring where the execution is verified from a log after the program finishes.

The rest of the paper is structured as follows. Section 2 overviews Quantified Event Automata. Section 3 introduces preliminaries. Section 4 introduces the models of execution and attacker we consider. 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 [7] (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 EA is built from a set of event names $\mathcal{N}$, a set of values $\mathtt{Val}$, and a set of variables $\mathtt{Var}$ (disjoint from $\mathtt{Val}$). The set of symbols is defined as $\mathtt{Sym} = \mathtt{Val} \cup \mathtt{Var}$. An event is a tuple $\langle \mathtt{e}, \mathtt{p}_1, \ldots, \mathtt{p}_n \rangle$, where $\mathtt{e} \in \mathcal{N}$ is the event name and $\mathtt{p}_1, \ldots, \mathtt{p}_n \in \mathtt{Sym}^n$ are the event parameters. We use a functional notation to denote events: $\langle \mathtt{e}, \mathtt{p}_1, \ldots, \mathtt{p}_n \rangle$ is denoted by $\mathtt{e}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$. Events that are variable-free are called ground events, *i.e.,* an event $\mathtt{e}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ is ground if $\mathtt{p}_1, \ldots, \mathtt{p}_n \in \mathtt{Val}^n$. A *trace* is defined as a finite sequence of ground events. We denote the empty trace by ϵ.

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 $\mathtt{True}$ 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 they 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 $\mathtt{M}$ with quantified variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$. We use the functional notation $\mathtt{M}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ to refer to the related EAs depending on the values taken by $\mathtt{x}_1, \ldots, \mathtt{x}_n$.

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-failure, *i.e.,* if no transition can be taken then there is an implicit transition to a (sink) failure state. Circular 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 $\frac{guard}{assignment}$ to write guards and assignments on transitions, := for variable assignment, and == for equality test.

Example 1 (QEA)**.** *Figure 1 depicts* $\mathtt{M_R}$*, a QEA that checks whether a given trace satisfies requirement* $\mathtt{R}$*: "for all* $\mathtt{i}$*, event* $\mathtt{e}_1(\mathtt{i})$ *should precede event* $\mathtt{e}_2(\mathtt{i})$*". The alphabet of* $\mathtt{M_R}$ *is* $\Sigma_{\mathtt{R}} = \{\mathtt{e}_1(\mathtt{i}), \mathtt{e}_2(\mathtt{i})\}$*. Consequently, any event in an input trace that is not an instantiation of* $\mathtt{e}_1(\mathtt{i})$ *or* $\mathtt{e}_2(\mathtt{i})$ *is ignored by* $\mathtt{M_R}$*. QEA* $\mathtt{M_R}$ *has two states (1) and (2), which are both accepting states, one quantified variable* $\mathtt{i}$*, and zero free variables. As its initial state (1) is an accepting state, the empty trace is accepted by* $\mathtt{M_R}$*. State (1) is a square state, hence an event* $\mathtt{e}_2(\mathtt{i})$ *at state (1) leads into an implicit failure state, which is what we want as it would not be preceded by event* $\mathtt{e}_1(\mathtt{i})$ *in this case. An event* $\mathtt{e}_1(\mathtt{i})$ *at state (1) leads into state (2) which is a skipping state, so after an occurrence of event* $\mathtt{e}_1(\mathtt{i})$ *any sequence of events (for the same value of* $\mathtt{i}$*) is accepted. We note that one can equivalently replace state (2) with an accepting square state with a self-loop labeled by* $\Sigma_{\mathtt{R}}$*.*

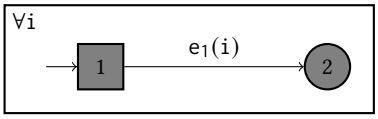

Figure 1: $\mathtt{M_R}$**, a QEA for requirement** $\mathtt{R}$ **from Example 1.**

Quantification $\forall\mathtt{i}$ *means that the property must hold for all values that* $\mathtt{i}$ *takes in a trace. Each instantiation of* $\mathtt{i}$ *results in an EA.*

To decide whether a trace is accepted by $\mathtt{M_R}$ *or not, the trace is first sliced based on the values that can match* $\mathtt{i}$*. Then, each slice is checked against the event automaton instantiated with the appropriate value for* $\mathtt{i}$*. For instance, the trace* $\mathtt{e}_1(\mathtt{I}_1).\mathtt{e}_2(\mathtt{I}_2).\mathtt{e}_2(\mathtt{I}_1).\mathtt{e}_1(\mathtt{I}_2)$ *is sliced into the following two slices:* $\mathtt{i} \mapsto \mathtt{I}_1 : \mathtt{e}_1(\mathtt{I}_1).\mathtt{e}_2(\mathtt{I}_1)$*, and* $\mathtt{i} \mapsto \mathtt{I}_2 : \mathtt{e}_2(\mathtt{I}_2).\mathtt{e}_1(\mathtt{I}_2)$*. Each slice is checked independently. The slice associated with* $\mathtt{I}_1$ *is accepted by* $\mathtt{M_R}(\mathtt{I}_1)$ *as it ends in the accepting state (2), while the slice associated with* $\mathtt{I}_2$ *is rejected by* $\mathtt{M_R}(\mathtt{I}_2)$ *since the event* $\mathtt{e}_2(\mathtt{I}_2)$ *at state (1) leads to an implicit failure state. Therefore, the whole trace is rejected by the QEA because of the universal quantification on* $\mathtt{i}$*.*

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

[1]The code is inspired from the C version in the FISSC benchmark [21].

Listing 1: Code of `verifyPIN`.

```
void verifyPIN() {
 g_authenticated = BOOL_FALSE;
 if(g_ptc > 0) {
  if(byteArrayCompare(g_userPin, g_cardPin) == BOOL_TRUE)
     {
   g_ptc = 3;      /*reset the counter of trials*/
   g_authenticated = BOOL_TRUE; }
  else
   { g_ptc--; }    /*one trial less remaining*/
 }
 return g_authenticated;
}
```

`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

- 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'.

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
 g_ptc := 3
 g_authenticated := BOOL_TRUE
 goto L1
 L2: g_ptc := -g_ptc
 L1: return g_authenticated
```

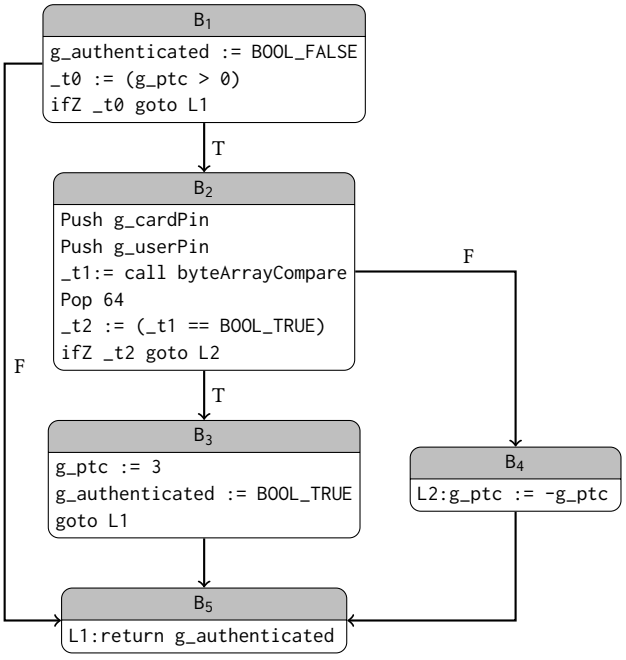

Figure 2: CFG for the `verifyPIN` function.

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 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`.

Example 4 (Test). *Function* `verifyPIN` *contains two tests:*

- `_t0 := (g_ptc > 0)`, `ifZ _t0 goto L1`
- `_t2 := (_t1 == BOOL_TRUE)`, `ifZ _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)` = `True` (*resp.* `(x oprel y)` = `False`) if B ends with a test `A` where `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 $\mathtt{e}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$, where $\mathtt{e}$ is the event name, and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ 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 $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ is the CFG of the program:

- For every basic block $\mathtt{B} \in \mathtt{V}$, event $\mathtt{begin}(\mathtt{i})$ has to be emitted at the beginning of $\mathtt{B}$.
- For every basic block $\mathtt{B} \in \mathtt{V}$, event $\mathtt{end}(\mathtt{i})$ has to be emitted:
  - just before instruction `return`, if the tail of $\mathtt{B}$ is `return`.
  - at the beginning of $\mathtt{B}_\mathtt{L}$, if the tail of $\mathtt{B}$ is an unconditional jump instruction `goto L`. Note that, in this case, events $\mathtt{end}(\mathtt{i})$ have to be emitted before event $\mathtt{begin}(\mathtt{i}_\mathtt{L})$.
  - at the beginning of both $\mathtt{B}_\mathtt{T}$ and $\mathtt{B}_\mathtt{F}$, if the tail of $\mathtt{B}$ is a conditional jump instruction `ifZ z goto L`. Note again that, in this case, events $\mathtt{end}(\mathtt{i})$ have to be emitted before events $\mathtt{begin}(\mathtt{i}_\mathtt{T})$ and $\mathtt{begin}(\mathtt{i}_\mathtt{F})$.
  - at the end of $\mathtt{B}$, otherwise.
- For every loop $\mathtt{B}_1, \ldots, \mathtt{B}_n$ in graph $\mathtt{G}$, events $\mathtt{reset}(\mathtt{i}_1), \ldots, \mathtt{reset}(\mathtt{i}_n)$ have to be emitted at the end of $\mathtt{B}_n$. Event $\mathtt{reset}(\mathtt{i})$ means that the basic block whose identifier is $\mathtt{i}$ may be executed again as it is involved in a loop. Note that, in this case, `reset` events have to be emitted before event $\mathtt{end}(\mathtt{i}_n)$ as the latter is used to detect jump attacks.
- For every basic block $\mathtt{B} \in \mathtt{V}$ that ends with a test $\mathtt{A}$, events $\mathtt{bT}(\mathtt{i}, \mathtt{x}, \mathtt{y})$ and $\mathtt{bF}(\mathtt{i}, \mathtt{x}, \mathtt{y})$ have to be emitted at the beginning of $\mathtt{B}_\mathtt{T}$ and $\mathtt{B}_\mathtt{F}$, respectively, where $\mathtt{cond}(\mathtt{A}) = (\mathtt{x}\ \mathtt{oprel}^\mathtt{i}\ \mathtt{y})$. We note that, in this case, the identifier $\mathtt{i}$ also identifies the test $\mathtt{A}$ and the logical operator $\mathtt{oprel}^\mathtt{i}$ as a basic block can contain at most one test.

We define a program execution as follows.

Definition 1. **(Program Execution).** *Let* $\mathtt{P}$ *be a program. An execution* $\mathtt{P}_\mathtt{exec}$ *of* $\mathtt{P}$ *is a finite sequence of events* $\mathtt{e}_1 . \cdots . \mathtt{e}_n$*, where* $\mathtt{n} \in \mathbb{N}$*, such that* $\mathtt{e}_\mathtt{j} \in \Sigma_{\mathtt{ALL}} = \{\mathtt{begin}(\mathtt{i}), \mathtt{end}(\mathtt{i}), \mathtt{reset}(\mathtt{i}), \mathtt{bT}(\mathtt{i}, \mathtt{x}, \mathtt{y}), \mathtt{bF}(\mathtt{i}, \mathtt{x}, \mathtt{y})\}$ *for every* $\mathtt{j} \in \{1, \ldots, \mathtt{n}\}$.

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

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 [5], 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 $\mathtt{B}$ into a basic block $\mathtt{B}'$ that is directly followed by a jump from $\mathtt{B}'$ into $\mathtt{B}$ 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 $\mathtt{B}$ that ends with a test $\mathtt{A} = \mathtt{z} := (\mathtt{x}\ \mathtt{oprel}\ \mathtt{y})$, `ifZ z goto L`. There is a test inversion attack on $\mathtt{A}$ when $\mathtt{B}_\mathtt{T}$ is executed when $(\mathtt{x}\ \mathtt{oprel}\ \mathtt{y}) = \mathtt{False}$, or when $\mathtt{B}_\mathtt{F}$ is executed when $(\mathtt{x}\ \mathtt{oprel}\ \mathtt{y}) = \mathtt{True}$. In practice, the result of $\mathtt{A}$ can be inverted, for example, by:

- skipping the conditional branch instruction, so that $\mathtt{B}_\mathtt{T}$ is executed regardless whether $(\mathtt{x}\ \mathtt{oprel}\ \mathtt{y})$ evaluates to `True` or `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* $\mathtt{P}$ *be a program, and let* $\mathtt{P}_\mathtt{exec} = \mathtt{e}_1, \ldots, \mathtt{e}_n$ *be an execution of* $\mathtt{P}$*. We say that there is a test inversion attack on* $\mathtt{P}_\mathtt{exec}$ *if it violates* $\mathtt{R}_1$ *or* $\mathtt{R}_2$ *which are defined as follows, where* $\mathtt{i}$ *identifies* $\mathtt{oprel}^\mathtt{i}$ *:*

- $\mathtt{R}_1$*: for every* $\mathtt{j}$*, if* $\mathtt{e}_\mathtt{j} = \mathtt{eT}(\mathtt{i}, \mathtt{x}, \mathtt{y})$ *then* $(\mathtt{x}\ \mathtt{oprel}^\mathtt{i}\ \mathtt{y}) = \mathtt{True}$.
- $\mathtt{R}_2$*: for every* $\mathtt{j}$*, if* $\mathtt{e}_\mathtt{j} = \mathtt{eF}(\mathtt{i}, \mathtt{x}, \mathtt{y})$ *then* $(\mathtt{x}\ \mathtt{oprel}^\mathtt{i}\ \mathtt{y}) = \mathtt{False}$.

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

- $\mathtt{tr}_1 = \mathtt{begin}(\mathtt{i}).\mathtt{begin}(\mathtt{i}).\mathtt{end}(\mathtt{i}).\mathtt{end}(\mathtt{i})$,
- $\mathtt{tr}_2 = \mathtt{begin}(\mathtt{i}).\mathtt{end}(\mathtt{i}).\mathtt{end}(\mathtt{i})$,
- $\mathtt{tr}_3 = \mathtt{begin}(\mathtt{i}).\mathtt{begin}(\mathtt{i}).\mathtt{end}(\mathtt{i})$, or
- $\mathtt{tr}_4 = \mathtt{begin}(\mathtt{i}).\mathtt{end}(\mathtt{i})$.

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 reset(i) should be emitted.

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

- R_3*: for every* j, *if* $e_j = begin(i)$ *then*
  - $e_{j+1} = end(i)$, *if* $e_{j-1} = begin(i)$.
  - $e_{j+1} = end(i)$, *or* $e_{j+1} = begin(i)$ *and* $e_{j+2} = end(i)$, *if* $e_{j-1} \neq begin(i)$.
- R_4*: for every* j, *if* $e_j = end(i)$ *then*
  - $e_{j-1} = begin(i)$, *if* $e_{j+1} = end(i)$.
  - $e_{j-1} = begin(i)$, *or* $e_{j-1} = end(i)$ *and* $e_{j-2} = begin(i)$, *if* $e_{j+1} \neq end(i)$.
- R_5*: there is no* j *such that* $e_j = end(i)$ *and* $e_{j+1} = begin(i)$.

Definition 3 considers the jump attacks that result in executions that cannot be built by concatenating elements from $\{tr_1, tr_2, tr_3, tr_4, reset(i)\}$. 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 begin(i) that is not directly followed by event 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 end(i) that is not directly preceded by event 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 end(i) that is directly followed by event 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 $P^{J}_{exec}(i)$ 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, $P^{J}_{exec}(i')$ 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* bT(i, x, y) *events and 4* bF(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* begin(i) *events and 14* end(i) *events) since* verifyPIN *has 5 basic blocks, contains two conditional branches, and every event has to be emitted twice.*

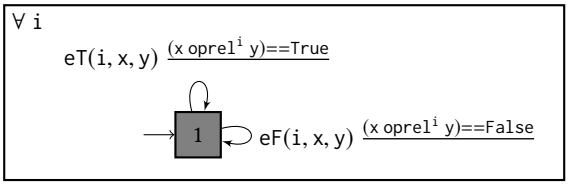

Figure 3: M_{TI}, a QEA detecting test inversion attacks.

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_{TI}, a QEA that detects test inversion attacks on a given execution. The alphabet of M_{TI} is $\Sigma_{TI} = \{eT(i, x, y), eF(i, x, y)\}$. Monitor M_{TI} has only one state, which is an accepting square state. It fails when event eT(i, x, y) is emitted while (x $oprel^i$ y) = False (*i.e.,* if the requirement R_1 is violated), or when event eF(i, x, y) is emitted while (x $oprel^i$ y) = True (*i.e.,* if the requirement R_2 is violated). M_{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 $oprel^i$ and it allows reporting the test that has been inverted in case of failure.

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}.

Example 6 (Test Inversion Attack)**.** *An attacker can perform a test inversion attack on* verifyPIN *by skipping the second conditional branch:* ifZ _t2 goto L2 *(Line 10 of Listing 2). This attack allows the attacker to get authenticated with a wrong PIN. Assuming a wrong PIN, function* byteArrayCompare *returns a value* BOOL_FALSE *(which 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* B_3 *(the success branch) instead of* B_4 *(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* Σ_{IT} *is as follows, where* I_j *is the identifier of* B_j*, 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* B_2*, the success branch of the first test. It takes the arguments* I_1*, 3, and 0 since the corresponding test is inside* B_1*, 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* B_3*, the success branch of the second*

[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_{TI} in this case, see Figure 3.

test. It takes the arguments I_2, BOOL_FALSE, *and* BOOL_TRUE *since the corresponding test is inside* B_2, *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_{IT}, *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 $i \mapsto I_1$ *satisfies both requirements* R_1 *and* R_2, *and thus it is accepted by* $M_{TI}(I_1)$. *While, slice* $i \mapsto I_2$ *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* $M_{IT}(I_2)$. *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* $i \mapsto I_2$ *is rejected by* $M_{IT}(I_2)$, *the whole faulted execution is rejected by* M_{IT}.

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.

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 Σ_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 (3) 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_J *is as follows, where* I_j *is the identifier of* B_j*:*

$begin(I_1).begin(I_1).end(I_1).end(I_1).\mathbf{begin(I_2)}.\mathbf{begin(I_2)}.begin(I_3)$
$.begin(I_3).end(I_3).end(I_3).begin(I_5).begin(I_5).end(I_5).end(I_5)$

The faulted execution is sliced by M_J, *based on the values that* i *can take, into the following four slices:*

$i \mapsto I_1 : begin(I_1).begin(I_1).end(I_1).end(I_1)$,

$i \mapsto I_2 : begin(I_2).begin(I_2)$,

$i \mapsto I_3 : begin(I_3).begin(I_3).end(I_3).end(I_3)$, *and*

$i \mapsto I_5 : begin(I_5).begin(I_5).end(I_5).end(I_5)$.

Slices $i \mapsto I_1$, $i \mapsto I_3$, *and* $i \mapsto I_5$ *satisfy requirements* R_3, R_4 *and* R_5. *Thus, they are respectively accepted by* $M_J(I_1)$, $M_J(I_3)$, *and* $M_J(I_5)$. *However, slice* $i \mapsto I_2$ *does not satisfy the requirement* R_3 *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* $M_J(I_2)$. *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,* $M_J(I_2)$ *ends in state (3), which is a failure state. Therefore, since slice* $i \mapsto I_2$ *is rejected by* $M_J(I_2)$, *the whole faulted execution is rejected by* M_J.

Note, given a CFG (V, E), monitor M_J 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 $M_J(i)$ 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_{TI}. 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 oprel y), it updates the currentState. If the monitor is in state Ok, the state is updated into Error if the received event is eT 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:

[4] www.eclipse.org/aspectj/

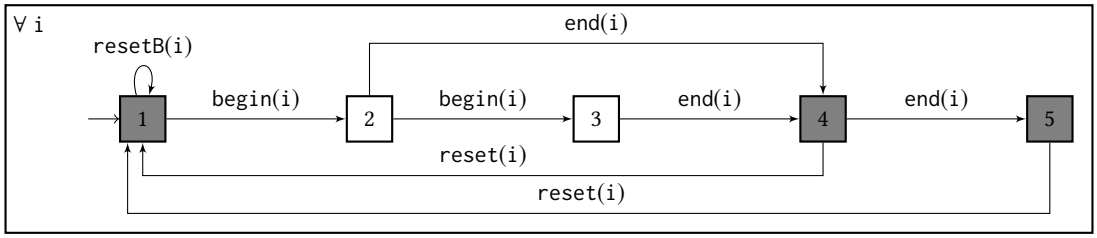

Figure 4: M_J, a QEA detecting jump attacks.

Listing 3: Java implementation of M_{TI}.

```
1  public class VerificationMonitorTI {
2   private State currentState = State.Ok;
3
4   public void updateState(Event e) {
5    switch (this.currentState) {
6     case Ok:
7      int x = e.getX();
8      int y = e.getY();
9      String oprel = e.getOprel();
10     boolean condition = evaluateCond(x,y,oprel);
11     if ((e.getName().equals("eT") && !condition) || (e.
           getName().equals("eF") && condition))
12      { this.currentState = State.Error; }
13     break;
14    case Error:
15     // No need to execute any code.
16     break;
17   }
18   System.out.println("moved to "+ this.currentState);
19  }
20
21  public Verdict currentVerdict () {
22   switch(this.currentState) {
23    case Ok: return Verdict.CURRENTLY_TRUE;
24    case Error: return Verdict.FALSE;
25    default: return Verdict.FALSE;
26   }
27  }
28 }
```

- P_{exec_1} 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` ≤ 0.
- P_{exec_2} which contains 20 events: 8 events `begin`, 8 events `end`, 2 events `eT` and 2 events `eF` that result from executing B_1, B_2, B_4 and B_5. This execution is performed when `g_ptc` > 0 and `g_userPin` $\neq$ `g_cardPin`.
- P_{exec_3} which contains 20 events: 8 events `begin`, 8 events `end` and 4 events `eT` that result from executing B_1, B_2, B_3 and B_5. This execution is performed when `g_ptc` > 0 and `g_userPin` = `g_cardPin`.

Table 1 summarizes the cumulative execution time and the memory footprint of 100K runs of P_{exec_3}: (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.[5]

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 [36] to directly instrument the bytecode would result in a smaller overhead.

Table 1: Cumulative execution time and memory footprint of 100K runs of P_{exec_3}.

	CPU Time (ms)	Memory (KB)
`verifyPIN`	164	284
Inst. `verifyPIN`	209 (× 1.27)	796.6 (× 2.8)
Inst. `verifyPIN` & Monitors	228 (× 1.39)	801 (× 2.82)

6.2 Test Inversion Attack

Function `verifyPIN` contains two tests. The first test: `_t0 := (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: `_t2 := (_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_icmpeq`, which performs a branch if the operands are equal.

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

[5]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.

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 stackmap 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 B_3. That is, after receiving 9 events: 4 events begin, 3 events end and 2 events eT. Note that the execution of B_2 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.

[6]Starting from Java 7, the typing system requires a stack map frame at the beginning of each basic block [40]. 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_icmpeq 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 byteArrayCompare. However, one can simulate the effect of goto using break and continue statements.

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 [19], monitoring IT logs [11], monitoring electronic exams [35], monitoring smart homes [22].

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 [3, 5, 15, 18, 32] such as computing a cryptographic operation twice then comparing the outputs. There are also approaches that use parity codes [34] 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, [24, 42, 49] use watchdogs for error detection at runtime. The underlying principle is to provide a watchdog processor [41] 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 [42], by verifying that the inserted assertions are executed in the order specified by the CFG. Ersoz *et al.* [24] propose the watchdog task, a software abstraction of the watchdog processor, to detect execution errors using some assertions. Saxena *et al.* [49] 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.* [45] 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 [12, 50]. Note that experiments have shown that SIED cannot detect all bit-flip faults. Later in [46], 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.* [51] 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.* [17] 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.* [29] 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.* [38] 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 [3, 23] or skipping the critical parts of the code [12, 50]. 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 [9]. 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 $\mathtt{M_{TI}}$ 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 [6, 8] 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 [44, 47]. 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 [6, 44]. 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 complier. 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.* [1], 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 [2, 48], and return-oriented attacks [52]. A related technique called control flow locking (CFL) has been introduced by Bletsch *et al.* [14] 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 [36] or JNIF [43], 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 [25, 27, 28] framework where some corrective actions and countermeasures are automatically executed and taken respecticely once an attack is detected.

Acknowledgment. The authors warmly thank the reviewers for their helpful comments on an earlier version of this paper. This work is supported by the French national program "Programme Investissements d'Avenir IRT Nanoelec" (ANR-10-AIRT- 05).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4:1–4:40. https://doi.org/10.1145/1609956.1609960

[2] William Arthur, Ben Mehne, Reetuparna Das, and Todd M. Austin. 2015. Getting in control of your control flow with control-data isolation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle

Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, 79–90. https://doi.org/10.1109/CGO.2015.7054189

[3] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. 2002. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In *Cryptographic Hardware and Embedded Systems - CHES 2002, Redwood Shores, USA, 2002, Revised Papers*. 260–275.

[4] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2011. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *FDTC'11, Tokyo, Japan, September 29, 2011*. 105–114.

[5] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2 (2006), 370–382. https://doi.org/10.1109/JPROC.2005.862424

[6] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*. ACM, 7. https://doi.org/10.1145/1873548.1873555

[7] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. 2012. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 68–84. https://doi.org/10.1007/978-3-642-32759-9_9

[8] Thierno Barry, Damien Couroussé, and Bruno Robisson. 2016. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016*.

[9] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2017. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer* (2017), 1–40. https://doi.org/10.1007/s10009-017-0454-5

[10] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*.

[11] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. 2014. Scalable Offline Monitoring. In *Runtime Verification: 5th International Conference, RV 2014. Proceedings*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer International Publishing, 31–47. https://doi.org/10.1007/978-3-319-11164-3_4

[12] Alberto Battistello and Christophe Giraud. 2015. Fault Cryptanalysis of CHES 2014 Symmetric Infective Countermeasure. *IACR Cryptology ePrint Archive* 2015 (2015), 500. http://eprint.iacr.org/2015/500

[13] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-François Lalande. 2012. High Level Model of Control Flow Attacks for Smart Card Functional Security. In *ARES 2012, Czech Republic, 2012*.

[14] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *ACSAC'11, Orlando, 2011*.

[15] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, 1997, Proceeding*.

[16] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 2001. On the Importance of Eliminating Errors in Cryptographic Computations. *J. Cryptology* 14, 2 (2001).

[17] Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. 2013. Detecting Laser Fault Injection for Smart Cards Using Security Automata. In *SSCC 2013, India, 2013. Proceedings*.

[18] Mathieu Ciet and Marc Joye. 2005. Practical Fault Countermeasures for Chinese Remaindering Based RSA (Extended Abstract). In *IN PROC. FDTC'05*. 124–131.

[19] Christian Colombo and Gordon J. Pace. 2013. Fast-Forward Runtime Monitoring — An Industrial Case Study. In *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 214–228. https://doi.org/10.1007/978-3-642-35632-2_22

[20] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. 2013. Electromagnetic Glitch on the AES Round Counter. In *COSADE'13, Revised Selected Papers*. 17–31.

[21] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Aude Crohen, and Philippe De Choudens. 2016. FISSC: a Fault Injection and Simulation Secure Collection. In *International Conference on Computer Safety, reliability and Security (LNCS)*, Vol. 9922. Springer Berlin / Heidelberg, Trondheim, Norway, 3–11. https://doi.org/10.1007/978-3-319-45477-1_1

[22] Antoine El-Hokayem and Yliès Falcone. 2018. Bringing Runtime Verification Home. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 222–240. https://doi.org/10.1007/978-3-030-03769-7_13

[23] Sho Endo, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki. 2014. A Multiple-Fault Injection Attack by Adaptive Timing Control Under Black-Box Conditions and a Countermeasure. In *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*. 214–228.

[24] A. Ersoz, D. M. Andrews, and E. J. McCluskey. 1985. *The watchdog task: Concurrent error detection using assertions*. Technical Report CRC – 85-8. Stanford University.

[25] Yliès Falcone. 2010. You Should Better Enforce Than Verify. In *Runtime Verification - First International Conference, RV 2010. Proceedings (Lecture Notes in Computer Science)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.), Vol. 6418. Springer, 89–105. https://doi.org/10.1007/978-3-642-16612-9_9

[26] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*. 141–175.

[27] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. 2018. Runtime Failure Prevention and Reaction. In *Lectures on Runtime Verification - Introductory and Advanced Topics*. 103–134. https://doi.org/10.1007/978-3-319-75632-5_4

[28] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design* 38, 3 (2011), 223–262. https://doi.org/10.1007/s10703-011-0114-4

[29] Arnaud Fontaine, Pierre Chifflier, and Thomas Coudray. 2015. PICON : Control Flow Integrity on LLVM IR. *SSTIC* (2015).

[30] Laurie Genelle, Christophe Giraud, and Emmanuel Prouff. 2009. Securing AES Implementation against Fault Attacks. In *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, 2009*.

[31] Klaus Havelund and Allen Goldberg. 2005. Verify Your Runs. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions (Lecture Notes in Computer Science)*, Bertrand Meyer and Jim Woodcock (Eds.), Vol. 4171. Springer, 374–383. https://doi.org/10.1007/978-3-540-69149-5_40

[32] Burton S. Kaliski Jr. and Matthew J. B. Robshaw. 1997. Comments on some new attacks on cryptographic devices. *RSA Laboratories Bulletin 5* (July 1997).

[33] Dusko Karaklajic, Jörn-Marc Schmidt, and Ingrid Verbauwhede. 2013. Hardware Designer's Guide to Fault Attacks. *IEEE Trans. VLSI Syst.* 21, 12 (2013).

[34] Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. 2003. Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers. In *CHES'03, Proceedings*.

[35] Ali Kassem, Yliès Falcone, and Pascal Lafourcade. 2017. Formal analysis and offline monitoring of electronic exams. *Formal Methods in System Design* 51, 1 (2017), 117–153. https://doi.org/10.1007/s10703-017-0280-0

[36] Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns.

[37] Dilip S. V. Kumar, Arthur Beckers, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. 2018. An In-Depth and Black-Box Characterization of the Effects of Laser Pulses on ATmega328P. In *CARDIS 2018, France, 2018, Revised Selected Papers*.

[38] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. 2014. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *Computer Security - ESORICS 2014, Poland, 2014. Proceedings, Part II*.

[39] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[40] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2013. The Java Virtual Machine Specification, Java SE 7 Edition.

[41] Davia J. Lu. 1980. Watchdog Processors and VLSI.

[42] Aamer Mahmood, Edward J. McCluskey, and Aydin Ersoz. 1985. Concurrent System-Level Error Detection Using a Watchdog Processor. In *Proceedings International Test Conference 1985, Philadelphia, PA, USA, November 1985*. 145–152.

[43] Luis Mastrangelo and Matthias Hauswirth. 2014. JNIF: Java Native Instrumentation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 194–199. https://doi.org/10.1145/2647508.2647516

[44] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. 2014. Formal verification of a software countermeasure against instruction skip attacks. *J. Cryptographic Engineering* 4, 3 (2014), 145–156. https://doi.org/10.1007/s13389-014-0077-7

[45] Bogdan Nicolescu, Yvon Savaria, and Raoul Velazco. 2003. SIED: Software Implemented Error Detection. In *DFT 2003, USA, Proceedings*.

[46] B. Nicolescu, Y. Savaria, and R. Velazco. 2004. Software detection mechanisms providing full coverage against single bit-flip faults. *IEEE Transactions on Nuclear Science* 51, 6 (Dec 2004).

[47] Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay. 2017. Fault Tolerant Infective Countermeasure for AES. *J. Hardware and Systems Security* 1, 1 (2017), 3–17. https://doi.org/10.1007/s41635-017-0006-1

[48] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware,*

and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings (Lecture Notes in Computer Science), Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.), Vol. 9148. Springer, 144–164. https://doi.org/10.1007/978-3-319-20550-2_8

[49] Nirmal R. Saxena and Edward J. McCluskey. 1990. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Trans. Computers* 39, 4 (1990).

[50] Jörn-Marc Schmidt and Christoph Herbst. 2008. A Practical Fault Attack on Square and Multiply. In *Fifth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2008, FDTC 2008, Washington, USA, 10 August 2008.*

[51] Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, and Jean-Louis Lanet. 2011. Evaluation of Countermeasures Against Fault Attacks on Smart Cards.

[52] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 552–561. https://doi.org/10.1145/1315245.1315313

[53] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. 2018. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *J. Hardware and Systems Security* 2, 2 (2018).

A PROOFS

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

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

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

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

Proof. Assume that $\mathtt{M_J}$ rejects $\mathsf{P_{exec}}$. We have to show that there is a jump attack on $\mathsf{P_{exec}}$, *i.e.,* there exists $\mathtt{i}$ such that $\mathsf{P^J_{exec}(i)} = \mathtt{e_1}, \ldots, \mathtt{e_n}$ violates $\mathsf{R_3}$, $\mathsf{R_4}$ or $\mathsf{R_5}$. As $\mathtt{M_J}$ rejects $\mathsf{P_{exec}}$ then there exists $\mathtt{i}$ such that EA $\mathtt{M_J(i)}$ fails while consuming $\mathsf{P^J_{exec}(i)}$. We split according to the cases in which $\mathtt{M_J(i)}$ fails.

- $\mathtt{M_J(i)}$ has encountered event $\mathtt{end(i)}$ in state (1). This means that there exists $\mathtt{j}$ such that $\mathtt{e_j} = \mathtt{end(i)}$, and that $\mathtt{e_{j-1}}$ is either ϵ or $\mathtt{reset(i)}$ since state (1) is the initial state and it can only be reached by $\mathtt{reset(i)}$. Thus, $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_4}$.
- $\mathtt{M_J(i)}$ ends in state (2). Then, in this case, there exists $\mathtt{j}$ such that $\mathtt{e_j} = \mathtt{begin(i)}$ since state (2) can only be reached from state (1) through event $\mathtt{begin(i)}$. Moreover, we can deduce that $\mathtt{e_{j+1}}$ is neither $\mathtt{begin(i)}$ nor $\mathtt{end(i)}$. Thus, $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_3}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{e_j} = \mathtt{reset(i)}$ in state (2). State (2) can only be reached from state (1) through event $\mathtt{begin(i)}$. Then, $\mathtt{e_{j-1}} = \mathtt{begin(i)}$. So, there exists $\mathtt{e_{j-1}} = \mathtt{begin(i)}$ such that $\mathtt{e_j}$ is neither $\mathtt{end(i)}$ nor $\mathtt{begin(i)}$ which violates $\mathsf{R_3}$.
- $\mathtt{M_J(i)}$ ends in state (3). Then there exists $\mathtt{j}$ such that $\mathtt{e_{j-1}} = \mathtt{e_j} = \mathtt{begin(i)}$ because state (2) can only be reached from state (1) and state (3) can only be reached from state (2), both through event $\mathtt{begin(i)}$. Moreover, $\mathtt{e_{j+1}} \neq \mathtt{end(i)}$ as there is a transition from state (3) into state (4) labeled by $\mathtt{end(i)}$, but $\mathtt{M_J(i)}$ ends in state (3). Thus, $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_3}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{reset(i)}$ in state (3). This case is similar to the previous case and violates $\mathsf{R_3}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{begin(i)}$ in state (3). Then there exists $\mathtt{j}$ such that $\mathtt{e_{j-1}} = \mathtt{e_j} = \mathtt{e_{j+1}} = \mathtt{begin(i)}$, which violates $\mathsf{R_3}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{begin(i)}$ in state (4). This means that there exists $\mathtt{j}$ such that $\mathtt{e_j} = \mathtt{begin(i)}$, and $\mathtt{e_{j-1}} = \mathtt{end(i)}$ since state (4) can only be reached by $\mathtt{end(i)}$ from state (2) or state (3), which can be reached only be reached by $\mathtt{begin(i)}$. Thus, $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_5}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{end(i)}$ in state (5). Then there exists $\mathtt{j}$ such that $\mathtt{e_{j-1}} = \mathtt{e_j} = \mathtt{e_{j+1}} = \mathtt{end(i)}$ because state (5) can only be reached from state (4) by $\mathtt{end(i)}$, and state (4) can only be reached by $\mathtt{end(i)}$. Thus, we have that $\mathtt{e_j} = \mathtt{e_{j+1}} = \mathtt{end(i)}$ and $\mathtt{e_{j-1}} \neq \mathtt{begin(i)}$, which violates $\mathsf{R_4}$.
- $\mathtt{M_J(i)}$ has encountered event $\mathtt{begin(i)}$ in state (5). This means that there exists $\mathtt{j}$ such that $\mathtt{e_j} = \mathtt{begin(i)}$ and $\mathtt{e_{j-1}} = \mathtt{end(i)}$ since state (5) can only be reached by $\mathtt{end(i)}$. Thus, $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_5}$.

Hence, there exists $\mathtt{i}$ such that $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_3}$, $\mathsf{R_4}$ or $\mathsf{R_5}$, and we can conclude about the first direction.

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

- If $\mathsf{P^J_{exec}(i)}$ violates $\mathsf{R_3}$, then it contains an event $\mathtt{e_j} = \mathtt{begin(i)}$ for some integer $\mathtt{j}$ such that:
  - $\mathtt{e_{j-1}} = \mathtt{begin(i)}$ and $\mathtt{e_{j+1}} \neq \mathtt{end(i)}$. In this case if, before receiving $\mathtt{e_{j-1}}$, $\mathtt{M_J(i)}$ was:
    - in state (1). Then $\mathtt{e_{j-1}} = \mathtt{begin(i)}$ leads into state (2) and $\mathtt{e_j} = \mathtt{begin(i)}$ leads into state (3). As $\mathtt{e_{j+1}} \neq \mathtt{end(i)}$ and from state (3) there is only one explicit transition labeled by $\mathtt{end(i)}$, then $\mathtt{M_J(i)}$ ends in state (3) (*i.e.,* $\mathtt{e_{j+1}} = \epsilon$) or a transition into an implicit failure state is fired.
    - in state (2). Then $\mathtt{e_{j-1}} = \mathtt{begin(i)}$ leads into state (3), and $\mathtt{e_j} = \mathtt{begin(i)}$ leads into an implicit failure state.
    - in state (3), (4) or (5). Then $\mathtt{M_J(i)}$ fails since none of the states (3), (4) and (5) has an explicit outgoing transition labeled by $\mathtt{begin(i)}$.
  - or $\mathtt{e_{j-1}} \neq \mathtt{begin(i)}$ and $\mathtt{e_{j+1}} \neq \mathtt{end(i)}$, and "$\mathtt{e_{j+1}} \neq \mathtt{begin(i)}$ or $\mathtt{e_{j+2}} \neq \mathtt{end(i)}$". This is equivalent to $\mathtt{e_{j-1}} \neq \mathtt{begin(i)}$ and, $\mathtt{e_{j+1}} = \epsilon$ (*i.e.,* $\mathtt{j} = \mathtt{n}$) or $\mathtt{e_{j+1}} = \mathtt{reset(i)}$ or "$\mathtt{e_{j+1}} = \mathtt{begin(i)}$ and $\mathtt{e_{j+2}} \neq \mathtt{end(i)}$" since $\Sigma_J = \{\mathtt{begin(i)}, \mathtt{end(i)}, \mathtt{reset(i)}\}$. In this case if, before receiving $\mathtt{e_{j-1}}$, $\mathtt{M_J(i)}$ was:

* in state (1). If e_{j-1} is end(i), a transition into an implicit failure state is fired. Otherwise, we have that e_{j-1} is reset(i) or ϵ (*i.e.,* $j = 1$), and thus $M_J(i)$ stays in state (1). Then, e_j = begin(i) leads into state (2). Then in state (2), if e_{j+1} is reset(i) a transition into an implicit failure state is fired; if e_{j+1} is ϵ then $M_J(i)$ ends in failure state (2); if e_{j+1} is begin(i) the transition from state (2) into state (3) is fired and, as $e_{j+2} \neq$ end(i) in this case, then a transition into an implicit failure state is fired or $M_J(i)$ ends in failure state (3).
* in state (2). If e_{j-1} is reset(i), a transition into an implicit failure state is fired. Otherwise, we have that e_{j-1} = end(i) which leads from state (2) into state (4). Then in state (4), as e_j = 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} is end(i), the transition into state (5) is fired. Then in state (5), as e_j = begin(i), a transition into an implicit failure state is fired. Otherwise, we have that e_{j-1} = reset(i), and thus the transition from state (4) into state (1) is fired. Then in state (1), as e_j = begin(i), the transition into state (2) is fired. Then in state (2), if e_{j+1} is reset(i) a transition into an implicit failure state is fired; if e_{j+1} is ϵ then $M_J(i)$ ends in state (2) which is a failure state; if e_{j+1} is begin(i) the transition from state (2) into state (3) is fired and, as $e_{j+2} \neq$ end(i) in this case, then a transition into an implicit failure state is fired or $M_J(i)$ ends in state (3), which is a failure state.
* in state (5). If e_{j-1} is reset(i) the reasoning is similar to the case of state (4) when e_{j-1} = reset(i). Otherwise, we have that e_{j-1} = end(i) which leads into an implicit failure state.

Hence, If $P^J_{exec}(i)$ violates R_3 then $M_J(i)$ fails, and thus M_J rejects P_{exec}.

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

  Hence, If $P^J_{exec}(i)$ violates R_4 then $M_J(i)$ fails, and thus M_J rejects P_{exec}.
- If $P^J_{exec}(i)$ violates R_5, then it contains an event e_j = end(i) for some integer j such that e_{j+1} = begin(i). In this case if, before receiving e_j, $M_J(i)$ was:
  - in state (1). Then event e_j = end(i) fires a transition into an implicit failure state.
  - in state (2). Then event e_j = end(i) fires the transition into state (4). In state (4), e_{j+1} = 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 = end(i) fires the transition into state (5). In state (5), e_{j+1} = begin(i) fires a transition into an implicit failure state.
  - in state (5). Then e_j = end(i) fires a transition into an implicit failure state.

  Hence, if $P^J_{exec}(i)$ violates R_5 then $M_J(i)$ fails, and thus M_J rejects P_{exec}.

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