AVFSM: A Framework for Identifying and Mitigating Vulnerabilities in FSMs

Adib Nahiyan[1], Kan Xiao[2], Kun Yang[1], Yier Jin[3], Domenic Forte[1] and Mark Tehranipoor[1]
[1]University of Florida; [2]University of Connecticut; [3] University of Central Florida
{adib1991, k.yang}@ufl.edu; kanxiao@engr.uconn.edu; yier.jin@eecs.ucf.edu; {dforte, tehranipoor}@ece.ufl.edu

ABSTRACT
A finite state machine (FSM) is responsible for controlling the overall functionality of most digital systems and, therefore, the security of the whole system can be compromised if there are vulnerabilities in the FSM. These vulnerabilities can be created by improper designs or by the synthesis tool which introduces additional don't-care states and transitions during the optimization and synthesis process. An attacker can utilize these vulnerabilities to perform fault injection attacks or insert malicious hardware modifications (Trojan) to gain unauthorized access to some specific states. To our knowledge, no systematic approaches have been proposed to analyze these vulnerabilities in FSM. In this paper, we develop a framework named Analyzing Vulnerabilities in FSM (AVFSM) which extracts the state transition graph (including the don't-care states and transitions) from a gate-level netlist using a novel Automatic Test Pattern Generation (ATPG) based approach and quantifies the vulnerabilities of the design to fault injection and hardware Trojan insertion. We demonstrate the applicability of the AVFSM framework by analyzing the vulnerabilities in the FSM of AES and RSA encryption module. We also propose a low-cost mitigation technique to make FSM more secure against these attacks.

1. INTRODUCTION
A major challenge associated with designing secure integrated circuits (ICs) is the diversity of existing and emerging attacks, attack goals, and potential countermeasures. It has been demonstrated that the security of cryptosystems, SoCs (system on chips) and micro-processor circuits can be compromised using timing analysis attacks [1], power analysis attacks [2], exploitation of design for test (DFT) structures [3], and fault injection attacks [4]. Also, due to the globalization of the semiconductor design and fabrication process, ICs are vulnerable to malicious modifications, referred to as hardware Trojans [5]. These hardware Trojans can create backdoors in the design through which sensitive information can be leaked and other possible attacks (e.g., denial of service, reduction in reliability, etc.) can be performed.

Current research is largely directed towards protecting the data path of the critical components against the aforementioned attacks. On the contrary, few work have focused on protecting the controller circuit of the device. Controller circuits are generally realized with a Finite State Machine (FSM) and the FSM is responsible for controlling the functionality of the whole system. The security of the overall system will be compromised if the FSM in the controller circuit is successfully attacked (e.g., by injecting fault or by Trojan insertion). In [6], authors have shown that the secret key of RSA encryption can be leaked by injecting fault into the FSM of the cryptographic device implementing the Montgomery ladder algorithm even if the data path is properly protected. To protect the FSM from fault injection attacks, concurrent error detection (CED) techniques (e.g., Triple Modular Redundancy, parity prediction, etc.) have been proposed in the literature [6]. These methods assume specific error models and thus will not work for other adversarial models [7]. Also, none of these methods consider the vulnerabilities introduced by the synthesis tool.

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2897992

In [8], authors have shown that the synthesis process of the FSM can introduce security risks in the implemented circuit by inserting additional don't-care states and transitions. In the RTL level of FSM there are don't-care conditions where the next state or the output of a transition are not specified. Logic synthesis tools take advantage of these don't-care conditions to optimize the design by introducing deterministic states and transitions for the don't-care conditions. The authors in [8] have shown that an attacker can utilize these don't-care states and transitions introduced by the synthesis process to implant hardware Trojans into the design and make the design untrusted. However, the authors did not specify an approach to evaluate the vulnerabilities introduced by these don't-care states and transitions.

There is a need for systematic approaches to identify vulnerabilities and security issues associated with ICs as early as possible in the design flow. The cost of fixing vulnerabilities found at later stages of design or post-silicon is significantly higher, following the well-known rule of ten. In addition, unlike software or firmware, there is little to no flexibility in changing or releasing post-shipment patches for hardware. Unfortunately, to our knowledge there are few efforts in place to systematically identify and quantify vulnerabilities present in ICs. In [9], the authors have presented a vulnerability analysis to determine a circuit's susceptibility to Trojan insertion at behavioral level. In [10], authors have proposed a leakage assessment methodology to evaluate the side channel vulnerability of an IC against power analysis attacks. The authors in [11] have proposed a metric to evaluate the vulnerability of a hardware structure to setup time violation based fault injection attacks. However, none of these techniques can be applied to FSM, as the vulnerability analysis of the FSM presents some unique challenges (e.g., existence of don't-care states and transitions) which we will discuss in the subsequent sections.

In this paper, we propose a framework, called Analyzing Vulnerabilities in FSM (AVFSM) to quantitatively analyze and evaluate how susceptible a FSM is against fault injection and/or Trojan attacks. To the best of our knowledge, this is the *first* systematic approach to analyze the vulnerabilities present in the FSM. We make the following major contributions:

- We propose a novel ATPG-based technique to extract the functionality of the FSM from a gate-level netlist. In addition to the states and transitions specified in the RTL level, our proposed technique can extract the don't-care states and transitions which are introduced during the synthesis process.
- Based on the extracted state transition table, our proposed framework, AVFSM analyzes each transition and reports all the transitions during which a fault can be injected to gain unauthorized access to some secured or protected states.
- We propose two metrics that use the extracted information as well as the design to quantify how susceptible the FSM is to fault injection and hardware Trojans. We analyze each reported transition using static timing analysis to determine the difficulty of inducing setup time violation based fault injection attacks.
- We perform case studies for the FSM of AES and RSA encryption modules using the proposed metrics. The results show that the vulnerability of the FSMs described above is a function of the state encoding scheme used. To our knowledge, this is a noteworthy observation that could impact future approaches to FSM design.
- We propose a low-cost mitigation technique to make the FSM secure against the aforementioned attacks. Our proposed

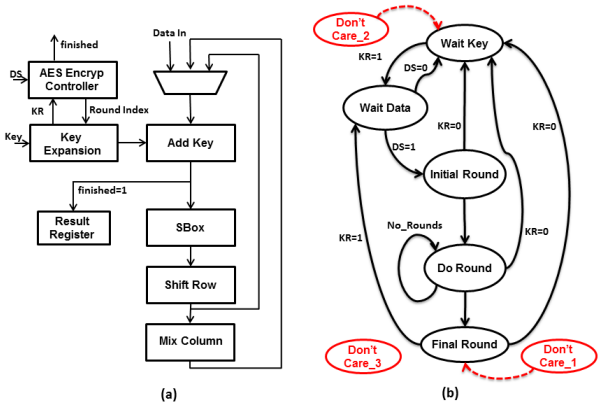

Figure 1: AES encryption module [12]. (a) AES encryption data path (b) FSM of the AES encrption controller. KR, DS stands for Key Ready and Data Stable signal, respectively.

technique replaces the state flip-flops (FFs) with programmable FFs that transfer the FSM into the reset state whenever a protected state is tried to be accessed from an unauthorized state.

The rest of the paper is organized as follows. In Section 2, we provide the necessary background on FSM. In Section 3, we elaborate on the possible attacks against FSM. We discuss our proposed AVFSM framework in Section 4. We present our results in Section 5 and propose a low cost mitigation approach in Section 6. We give our concluding remarks in Section 7.

2. PRELIMINARIES AND DEFINITIONS

An FSM is formally defined as a 5-tuple $(S, I, O, \varphi, \lambda)$, where S is a finite set of states, I is a finite set of input symbols, O is a finite set of output symbols, $\varphi : S \times I \rightarrow S$ is the next-state function and $\lambda : S \times I \rightarrow O$ is the output function.

For convenience, an FSM is typically represented as a directed graph where each vertex represents a state $s \in S$ and an edge represents the transition $t = T(x,y)$ from current state x to its next state y. This graph is referred to as a *state transition graph* (STG). In the STG each state can be accessed from a set of states which we define as the *accessible set of states*,

$$A(x) = \{y \mid y \text{ is accessible from } x\} \quad (1)$$

In this paper, we define two more sets, P and L, which are both specified by the designer. P is the set of protected states and L is a set of authorized states that are allowed access to a protected state p, that is $A(L) = \{p \mid p \in P\}$. If any state p is accessed by any state apart from states in L then the security of the FSM can be compromised.

In the behavioral specification of the FSM there are don't-care conditions where the next state or the output of a transition are not specified. During the synthesis process, the synthesis tool tries to optimize the design by introducing deterministic states and transitions for the don't-care conditions. Let us consider the FSM F' implemented by the synthesis tool from the behavioral description of the FSM F. Let, S and S' represent the set of states and T and T' represent the set of transitions in F and F', respectively. The set of don't-care states and transitions (S_D and T_D) introduced by the synthesis process are defined as follows,

$$S_D = \{s' \mid (s' \in S') \cap (s' \notin S)\};\ T_D = \{t' \mid (t' \in T') \cap (t' \notin T)\} \quad (2)$$

3. THREAT MODEL

In this section, we explore the possible attacks against FSM and show how these attacks can compromise the security of the overall system. We shall use the controller circuit of an AES encryption module [12] (see Fig. 1(b)) as an example to demonstrate the feasibility and effectiveness of these attacks.

The state transition diagram of the FSM shown in Fig. 1(b) implements the AES encryption algorithm on the data path shown in Fig. 1(a). The FSM is composed of 5 states: 'Wait Key', 'Wait Data', 'Initial Round', 'Do Round' and 'Final Round'. Each of these states controls specific modules during the ten rounds of AES encryption. After ten rounds, the 'Final Round' state is reached and the FSM generates the control signal $finished = 1$ and this signal stores the result of the 'Add Key' module (i.e., the ciphertext) in the 'Result Register'. For this FSM, we define 'Final Round' as a protected state because if an attacker can gain access to the 'Final Round' without going through the 'Do Round' state then premature results will be stored, potentially leaking the secret key. For example, if an attacker can get to 'Final Round' state from the 'Initial Round' state, then instead of the ciphertext, $key \oplus plaintext$ value will be stored in the 'Result Register' and the attacker can easily obtain the key of the AES encryption. Therefore, for this example, the set of protected states is $P = \{Final\ Round\}$ and the set of authorized states $L = \{Do\ Round\}$. Consider the states and transitions marked in red in the STG of Fig. 1(b) which represent don't-care states (S_D) and transitions (T_D) introduced by the synthesis tool.

We will consider the following attacks against the FSM:

Fault Injection Attack: This attack strategy relies on injecting a fault in the FSM during a specific transition that will cause the FSM to enter a protected state p through a state other than an authorized state in L. The attacker may also inject the fault to go to a don't-care state that has access to a protected state (e.g., in Fig. 1(b) an attacker can inject a fault to go to state 'Don't Care_1' and access the protected state 'Final Round' from this state). For the fault attack model, we assume the attacker is the end user and manipulates the clock signal, supply voltage, or operating temperature to inject such faults.

Trojan Attack: In this attack scenario, the attacker inserts illegal states or manipulate the STG so that the FSM will go to the protected state when a certain signal is triggered. While doing these malicious modifications, the attacker's objective is to design a hardware Trojan that occupies negligible portion of the overall circuit and has little effect on the power and timing of the original circuit so that the Trojan can evade the verification and validation testing. The presence of the don't-cares gives the attacker a unique advantage to insert Trojans that exploit these states and transitions. If a don't-care state has access to a protected state then it poses the most serious threat (e.g. the state 'Don't Care_1'in Fig. 1(b)), in which case the attacker only needs to add a few gates without changing the original FSM structure in order to go to the protected state from that don't-care state when the trigger signal is launched. Note that for this attack, we consider the attacker to be an in-house designer (i.e., rogue employee) or an untrusted foundry.

It is clear from the above description that the don't-care states and transitions inserted by the synthesis tool can lead to several major security vulnerabilities.

4. AVFSM FRAMEWORK

Our objective is to develop a comprehensive framework, called AVFSM, for automatically analyzing the vulnerability of FSMs against fault injection and Trojan attacks. The proposed framework will be the first, to the best of our knowledge, to address such vulnerabilities in the FSM. AVFSM takes as input (i) gate-level netlist of the design; (ii) FSM synthesis report; and (iii) user given inputs and then outputs the vulnerabilities present in the FSM. In this paper, we also tie the mentioned vulnerabilities to a set of metrics so that each FSM's design can be quantitatively analyzed.

The overall workflow of our AVFSM framework is shown in Fig. 2. The AVFSM framework is composed of four modules:

- FSM Extraction (FE): Extracts the STG of the FSM from the gate-level netlist.
- Don't-care S_D&T_D Identification ($DCSTI$): Reports the S_D and T_D introduced by the synthesis process.

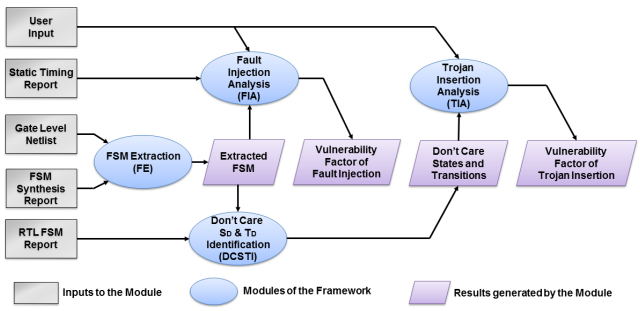

Figure 2: Overall workflow of the AVFSM framework.

Table 1: Symbols and notations

Symbols	Definitions	Symbols	Definitions
P	set of protected states	N_{VT}	total number of vulnerable transitions
L	set of authorized states	$Path_{FS}(i)$	maximum path delay of ith state FF
S_D	set of don't-care states	SF	susceptibility factor
T_D	set of don't-care transitions	PVT	percentage of vulnerable transitions
F_S	state flip-flop (FF)	ASF	average susceptibility factor
S_{EN}	state encoding	VF_{FI}	vulnerability factor of fault injection
VT	vulnerable transitions	VF_{Tro}	vulnerability factor of Trojan insertion

- Fault Injection Analysis (FIA): Reports a quantitative measure of the vulnerability of FSM to fault injection attack.
- Trojan Insertion Analysis (TIA): Reports a quantitative measure of the vulnerability of FSM to Trojan insertion.

The brief description of each module will be given in the subsequent sections. Table 1 presents the symbols and notations that we use throughout this paper.

4.1 Extraction of STG

To analyze vulnerabilities in the FSM, we first need to extract the *state transition graph* (STG) from the synthesized gate-level netlist. The extracted STG must incorporate the don't-care states and transitions which were introduced by the synthesis process. Existing work in literature only focuses on FSM reverse engineering from gate-level netlist [13], [14]. However, none of these techniques can extract the STG with the S_D and T_D.

One straightforward approach would be to perform a functional simulation of the FSM with all possible input patterns and produce the STG. However, this technique also cannot extract the don't-care states and transitions as these don't-care states cannot be accessed under the normal operating conditions of the FSM (see Fig. 1(b)). It is because of the fact that the synthesis tool introduces these don't-care states in such a way that these states cannot be accessed from the normal states (states mentioned in the RTL code); otherwise the original functionality of the FSM will be altered.

We propose an automatic test pattern generation (ATPG) based FSM extraction technique which can produce the STG with the don't-care state and transitions from the synthesized netlist. Our proposed extraction technique takes the gate-level netlist and the FSM synthesis report as inputs, and automatically generates the STG. Here, our assumption is that this vulnerability analysis will be performed by the designer, who has access to the RTL code, gate-level netlist, synthesis report and therefore has knowledge of the functionality of the FSM.

Module FE of our AVFSM framework is responsible for this extraction process. The algorithm of this module is shown in **Algorithm 1**. The algorithm includes two procedures; procedure I generates a modified netlist for the ATPG analysis and the procedure II extracts the STG of the FSM.

Procedure I first identifies the state flip-flops (F_S) using the FSM synthesis report generated by the synthesis tool (procedure I, line 4). In this work, we use the *dc_shell (Synopsys)* tool's *report_fsm* command to generate the report. The report contains names of the state registers (F_S) and the state encoding information (S_{EN}). The naming of the registers is conserved during the synthesis process and we can identify the state FFs using the FSM synthesis report.

After identifying the state FFs, our algorithm searches if there are any non-state FFs (f_{NS}) present in the input cone of the state FFs (see Fig. 3(a)) (procedure I, line 6). These non-state FFs are typically counters and they influence the state transitions of the FSM (e.g., in Fig. 1(b) the *No_Rounds* in *Do Round* state is counted with four counter FFs). We wish to determine the logic values of these non-state FFs which cause a transition in the STG. For example, when four counter FFs reach logic value of 1010, the state transition from *Do Round* to *Final Round* should take place.

Procedure I then generates the modified netlist for the ATPG analysis according to the steps shown in lines 5 to 16. The modified netlist is shown in Fig. 3(b) and the original FSM is shown in Fig. 3(a). In the modified netlist, the output nets of the state FFs (which define the present state) and the non-state FFs (which define conditions for state transition) are connected as primary inputs, $PI_{PresentState}$ and PI_{FNS}. Also, XOR gates are placed at each input net of F_s and the other input of the XOR gate is connected as primary input, PI_{XOR}. The output pins of the XOR gates are ORed together and the output pin of the OR gate is added as primary output, PO_{OR}. This modified netlist will be used by procedure II to generate STG of the FSM.

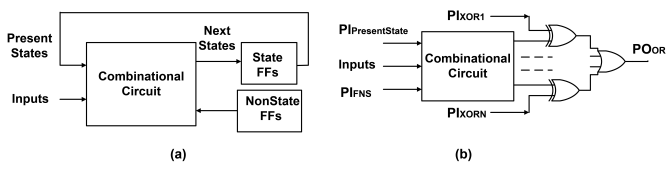

Figure 3: (a) Original FSM (b) Modified FSM for ATPG-based STG extraction.

Algorithm 1 Extraction of STG of FSM

```
1:  procedure I: MODIFIED NETLIST GENERATION
2:  Input: Gate-level netlist of the FSM, FSM synthesis report
3:  Output: Modified netlist for ATPG-based FSM extraction
4:  F_S ← Identify state FFs
5:      for each f ∈ F_S do
6:          f_NS ← Identify non-state FFs in the input cone of f
7:          Remove f_NS and add the net where Q pin of f_NS was connected as primary input, PI_FNS
8:          Add inverter to compensate for QN pin of f_NS
9:      end for
10:     for each f ∈ F_S do
11:         Remove f from the Netlist
12:         Add the net where Q pin of f was connected as primary input, PI_PresentState
13:         Add XOR gate at the net where D pin of f was connected
14:         Add the other input of XOR gate as primary input, PI_XOR
15:         ORed all the outputs of XOR gates and add output of the OR gate as primary output, PO_OR
16:         Add inverter to compensate for QN pin of f
17:     end for
18: end procedure
1:  procedure II: STATE TRANSITION GRAPH EXTRACTION
2:  Input: Modified gate-level netlist, FSM synthesis report
3:  Output: Extracted State Transition Graph
4:  S_EN ← Get state encodings
5:      for each s ∈ S_EN do
6:          Apply the logical value of s as constraint on PI_XOR
7:          Remove all faults and add stuck-at-1 fault at PO_OR
8:          Generate test patterns n times for the mentioned fault
9:          Extract the present state values and conditions that causes transition to s from the generated test patterns
10:     end for
11: end procedure
```

Procedure II determines the present states and input conditions which cause transition to a particular state $s \in S_{EN}$. The basic idea is to first apply the logical values of s as constraints on PI_{XOR} and generate test patterns for stuck-at-1 fault at PO_{OR} (procedure II, line 6-8). To generate patterns for this fault, the ATPG tool must produce 0 at PO_{OR} which requires the logic values at the input of the XOR gates to match with the constraints (s) applied on PI_{XOR}. In other words, the ATPG tool will generate the logic values of present states ($PI_{PresentState}$) and input conditions (i.e., input pins of the FSM and PI_{FNS}) which cause transitions to state s. We generate the test patterns n number of times using *Tetramax (Synopsys)* tool's *n-detect* option to get all possible present states and input conditions which cause transition to s. Although this option does not guarantee generation of all possible patterns for a specific fault, in our experiments we have verified that by specifying suitable value of n, we can extract the whole STG.

After Module FE extracts the STG from gate-level netlist, Module $DCSTI$ compares it to the STG from the RTL code and reports the additional don't-care states and transitions. There are commercial tools available which can extract the STG from RTL code. In this paper we have used *Altera's Quartus* tool for this purpose.

4.2 Vulnerability Analysis: Fault Injection

In this section, we use the extracted STG to analyze how susceptible the FSM is to fault injection attacks. In our analysis, we consider the faults which can be injected by violating the setup timing constraints using overclocking, voltage starving, and/or heating the device [15]. These types of attacks require low-cost equipment and pose the most serious threat. In this paper, we do not consider attackers with the capabilities to induce faults in one or more logic

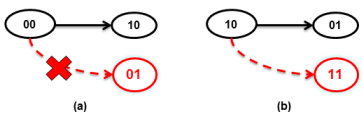

Figure 4: Setup time violation based fault injection attack. (a) and (b) are two examples where fault is not possible and possible respectively.

Algorithm 2 Conditions for fault attack

1: **procedure**
2: **Input:** Extracted State Transition Graph
3: **Input:** set of protected state P ▷ User given input
4: **Output:** Conditions for successful fault attack
5: $T(x,y) \leftarrow$ Extracted State Transition Graph
6: **for** each $T(x,y)$ **do** ▷ Transition from x to y
7: $S_x = [b_{x(n-1)}.....b_{x1}b_{x0}]$ ▷ S_x is state encoding of x
8: $S_y = [b_{y(n-1)}.....b_{y1}b_{y0}]$ ▷ b represents each bit of S
9: $P = [b_{p(n-1)}.....b_{p1}b_{p0}]$
10: **Compute** $C = \prod_{i=0}^{(n-1)}((b_{xi} \oplus b_{yi}) + (b_{xi}.b_{pi}))$
11: **if** $(C == 1)$ **then**
12: Fault attack possible for $T(x,y)$
13: $VT(x,y) \leftarrow T(x,y)$
14: **for** $i = 0$ *to* $(n-1)$ **do**
15: **if** $(b_{xi} \oplus b_{yi})$ **then**
16: **if** $(b_{xi} == b_{pi})$ **then**
17: $PathViolated_{x,y} = \{Path_{Fs}(i)\}$
18:
19: **else**
$PathOK_{x,y} = \{Path_{Fs}(i)\}$
20: **else**
21: $PathNoEffect_{x,y} = \{Path_{Fs}(i)\}$
22: **end for**
23: **else**
24: Fault attack not possible for $T(x,y)$
25: **end for**
26: **end procedure**

gates of a circuit with a precisely focused light beam.

Estimating the vulnerability of hardware cryptosystems against timing violation attacks have been recently proposed in [11]. However, their proposed technique can only be applied to the data path and not to the FSM. Unlike data path, the FSM presents some unique challenges in vulnerability analysis of fault injection attack (e.g., existence of don't-care states and transitions). Here, we propose a technique which analyzes each transition of the STG and based on a proposed metric quantitatively measures how susceptible that transition is to a fault injection attack. Based on the result, our AVFSM framework will automatically report overall vulnerability measures of the FSM to fault attack.

Our vulnerability analysis is based on the observation shown in Fig.4. Let us consider the state transition $T(00,10)$ where the current state is 00 and the next state is 10. During this transition, one cannot perform time violation based fault injection attack to go to state 01 (see Fig. 4(a)). It is because during this state transition ($T(00,10)$), the LSB bit of both the current state and the next state remains 0 and therefore, a setup time violation based fault cannot be injected at this bit position to change the bit value to 1. On the other hand during $T(10,01)$, one can inject a fault to go to state 11 (see Fig. 4(b)). To successfully inject this fault, the setup time constraint of MSB state FF needs to be violated whereas the setup time constraint of LSB state FF needs to be maintained. In other words, delay of the logic path of MSB state FF needs to be greater than the delay of the logic path of LSB state FF.

To perform the fault vulnerability analysis, Module FIA of our proposed AVFSM framework first reads the extracted STG and obtains the set of protected state P from the designer. Then the module looks into each state transition and analyze if a fault can be injected during this transition to gain access to a protected state. This analysis is performed according to **Algorithm 2**. The algorithm first computes the condition, C (line 10) for each transition and if $C == 1$ then it considers the respective transition as *Vulnerable Transition*, VT (lines 11-12). VT is defined as a set of transitions during which a fault can be injected to gain access to a protected state. For each VT, **Algorithm 2** reports the conditions that need to be satisfied to perform a setup time violation based fault attack which are shown below,

- $PathViolated$: setup time constraint of the state FF in this path needs to be violated (lines 16-17).
- $PathOK$: setup time constraint of the state FF in this path needs to be maintained (lines 18-19).
- $PathNoEffect$: state logic bit in this path does not change during the transition and therefore this path has no impact on the vulnerability analysis (lines 20-21).

Apart from the protected states, Module FIA also considers the don't-care sates that have access to the protected states (e.g., the Don't-care_1 state in Fig. 1(b)) and reports the transitions as VT which can give access to these don't-care states. These don't-care states are defined as *Dangerous Don't-Care States* (*DDCS*) and mathematically can be represented as,

$$DDCS = \{s' \mid (A(s') = P) \cap (s' \in S_D)\} \tag{3}$$

Now, each VT may not pose the same level of threat to the implemented FSM. To quantify how susceptible each VT is to fault injection attack, Module FIA uses *Synopsys's Primetime* tool (for static timing analysis (STA)) to get the maximum path delay of each state FFs. We propose the *susceptibility factor* metric, SF to quantitatively measure the vulnerability of each transition is to fault injection attack,

$$SF = \frac{Path_{Difference}}{avg(Path_{FS})} \tag{4}$$

SF is function of $Path_{Difference}$ where $Path_{Difference}$ is defined as,

$$\begin{aligned} &Path_{Difference} = Path_{FS}(i) - Path_{FS}(j) \\ &where\ i \in PathViolated\ and\ j \in PathOK \end{aligned} \tag{5}$$

Here, $Path_{Fs}(i)$ represents the maximum path delay to ith state FF and $avg(Path_{FS})$ is calculated by taking the mean value of all the $Path_{FS}$. Note that we consider the $Path_{FS}$ to be normally distributed with the mean value of maximum path delay (reported by STA) and a variance value of 5% of $avg(Path_{FS})$ to take into account of process variation.

Now, a high value of $Path_{Difference}$ between $PathViolated$ and $PathOK$ means that the attacker has a greater ability to successfully inject the fault. Therefore, a higher SF indicates that the respective VT is more susceptible to fault injection attack. On the other hand if SF is negative then it means delay of $PathOK$ is higher than delay of $PathViolated$ and fault injection for this transition is not feasible in the implemented circuit. Therefore, the transitions with negative SF is removed from the set of vulnerable transitions VT.

Module FIA use the metric, *vulnerability factor of fault injection* (VF_{FI}) to measure the overall vulnerability the FSM to fault injection attack. VF_{FI} is defined as follows

$$VF_{FI} = \{PVT(\%), ASF\}, where \tag{6}$$

$$PVT(\%) = \frac{Total_{Vulnerable\_Transition}(N_{VT})}{Total_{Transition}},\ ASF = \frac{\sum_{i=1}^{N_{VT}} SF(i)}{N_{VT}}$$

The metric VF_{FI} is composed of two parameters $\{PVT(\%), ASF\}$. $PVT(\%)$ indicates the percentage of VT to $Total_{Transition}$ and ASF signifies the average of SF. The greater the values of these two parameters are, the more susceptible the FSM is to fault attacks.

4.3 Vulnerability Analysis: Trojan Attack

In this section, we use the extracted STG to analyze how susceptible the FSM is to Trojan attacks. In our analysis, we consider the don't-care states which can be utilized to insert Trojans and gain access to a protected state.

During the synthesis process if a don't-care state is introduced that has direct access to a protected state (*DDCS*) then it can create a vulnerability in the FSM by allowing the attacker to utilize this don't-care state to insert a Trojan to gain access to the protected state. Let us consider the FSM of AES encryption module shown in Fig. 4(b). The 'Don't-Care_1' state is introduced by the synthesis tool and this state has direct access to the protected state 'Final Round'. An attacker needs only to add a small triggering circuit to gain access to the 'Don't-Care_1' and then go to 'Final Round' state from this state without changing the basic structure of the FSM. An attacker can thus bypass the intermediate rounds of AES encryption and get access to the key. From the attacker's point of view the existence of the 'Don't-Care_1' state presents a unique advantage to insert the Trojan with negligible area overhead. Also, the don't-care states are not part of verification and validation testing; therefore these Trojans are likely to evade detection.

In our proposed AVFSM framework, Module TIA performs the vulnerability analysis of Trojan attacks. It first reads the extracted STG and the don't-care states (S_D) and transitions reported by Module FE and Module $DCST1$, and gets the set of protected states (P) from the designer. Module TIA then searches for the *Dangerous Don't-Care States* (*DDCS*) and use the following metric *vulnerability factor of Trojan insertion* (VF_{Tro}) to evaluate the vulnerability of the FSM to Trojan insertion.

$$VF_{Tro} = \frac{Total\ number\ of\ s'}{Total_{Transition}}, where, s' \in DDCS \tag{7}$$

For a secure design, this metric's value should be zero. This metric enables the designer to evaluate the FSM's vulnerability to possible Trojan insertion which were introduced by the synthesis tool. *Note that the proposed vulnerability analysis only considers Trojans which exploit a don't-care state to perform the attack. There are other possible approaches to insert Trojans in the FSM (e.g. the specification of FSM itself can be maliciously modified) and vulnerability analysis of these attacks are out of scope.*

5. RESULTS

In this section, we first verify the correctness and scalability of our FSM extraction technique (Module FE). We then demonstrate, in details the applicability of the AVFSM framework by analyzing the vulnerabilities in the FSMs of AES and RSA module.

5.1 Result of FSM Extraction Technique

We apply our proposed FSM extraction technique to a number of FSM benchmark circuits from OpenCores [12]. These benchmark circuits are described in RTL code. We use Synopsys *Design Compiler* to get the synthesized gate-level netlist along with the FSM synthesis report. We then use Module FE to extract the STG from the synthesized gate-level netlist and use Module *DCSTI* to get the don't-care states and transitions. Table 2 demonstrates the details of each benchmark.

Table 2: Results of proposed FSM extraction technique.

Benchmark	State FFs	Input pins	RTL states	RTL transitions	Don't-care states	Don't-care transitions	CPU time (s)
Multiplier	3	10	5	8	3	3	< 0.05
AES Encryption	3	10	5	11	3	6	< 0.05
RSA Encryption	3	9	7	9	1	1	< 0.05
Prep4 (One Hot)	16	26	16	40	38	41	< 0.9
SAP Computer	4	10	12	25	4	4	< 0.1
UART	3	20	5	13	3	3	< 0.05
MIPSR 2000	5	19	19	33	10	10	< 0.2

The 'State FFs' column in Table 2 represents the number of state FFs used to implement the design. The 'Input pins' column represents the total number of input pins ($PI_{PresentState}$, PI_{FNS}, and *primary input pins of FSM*) in the modified gate-level netlist for which ATPG patterns are to be generated (see Section 4.1). The 'RTL States' and 'RTL Transitions' are obtained from the RTL STG which is generated using *Altera's Quartus* tool. The 'Don't-care states' and 'Don't-care transitions' are obtained by comparing the STG extracted from gate-level netlist and RTL STG. Note that, some don't-care states have no transition to other states in the FSM (e.g., the Don't-care_3 state in Fig. 1(b)). These don't-care states are not extracted by our AVFSM framework and they are not listed in the 'Don't-care states' column. The 'CPU time' column shows the time in seconds to generate all the ATPG patterns.

We first verify the efficacy of our proposed extraction technique by comparing the extracted STG with the RTL STG. For all the benchmark circuits except the AES encryption, we have found that $RTL\ STG \subseteq extracted\ STG$. That is, all the transitions and states in the RTL STG are present in the extracted STG. This observation verifies that our proposed FSM extraction technique can accurately extract the STG from the gate-level netlist. Only for the FSM of AES encryption circuit one transition in the RTL STG was not present in the extracted STG. One possible explanation for this mismatch is that the missing RTL transition could be a redundant condition and, therefore, was removed during the synthesis process.

The time required to generate the ATPG patterns depends on the total number of states, number of 'Input pins' and how many ATPG patterns are generated. Note that, we have used *Tetramax's n-detect* option to generate multiple patterns for a specific fault. We set $n = 1000$ for Prep4 benchmark and $n = 100$ for all the other benchmarks. The reason for using larger n value for the Prep4 is because this benchmark is encoded in 'One Hot' style where the number of state FFs equal to the number of states. Now, for all the benchmark circuits, the time required to generate the ATPG patterns is less than 1 second. The size of these benchmarks range from the small scale controller circuit of a multiplier to a medium scale controller circuit of a microprocessor (MIPSR 2000). By observing the time required to generate ATPG patterns from Table 2, we can conclude that our proposed FSM extraction technique is quite scalable.

It should be noted that the time shown in Table 2 does not incorporate the time needed to generate the modified netlist or STG from ATPG patterns. These operations include reading and creating text files and therefore, does not require significant amount of time.

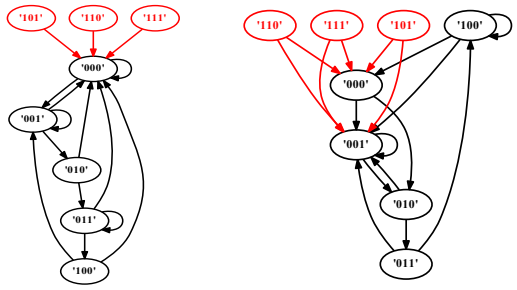

Figure 5: Extracted STG of (a) scheme 1 (left), (b) scheme 2 (right)

5.2 Case Study: I

We apply our proposed AVFSM framework to two implementations of AES encryption module's controller circuit [12] and compare each implementation's vulnerability. The data path and the FSM of the controller circuit of the AES encryption module is shown in Fig. 1(a) and (b). As discussed in Section 3, the attackers' objective would be to get to the 'Final Round' without going through the 'Do Round' stages in order to gain access to the key. Therefore, for this FSM, the set of protected states is $P = \{Final\ Round\}$ and the set of authorized states is $L = \{Do\ Round\}$.

For this case study, we have used two different encoding schemes for the FSM of the AES encryption module. We use { *Wait Key, Wait Data, Initial Round, Do Round, Final Round*} = { *000, 001, 010, 011, 100*} and { *Wait Key, Wait Data, Initial Round, Do Round, Final Roun*} = { *001, 010, 011, 100, 000*} for schemes 1 and 2, respectively. We then synthesize each scheme with medium area effort and apply our proposed AVFSM framework to analyze the vulnerabilities of each implemented FSM.

The first part of our analysis is to produce the STG from the gate-level netlist. The extracted STG of schemes 1 and 2 are shown in Fig. 5(a) and (b), respectively. The red colored states and transitions represent the don't-care states and transitions; whereas the black colored states and transitions represent the RTL states and transitions. Both schemes operate identically under normal operating condition. However, as we will show in the following section, one scheme is more vulnerable to faults and Trojan attacks than the other.

For the fault injection vulnerability analysis, AVFSM analyzes all transitions in the STG and reports the transitions which are vulnerable to fault attacks. For scheme 2, AVFSM reports 12 VT during which a fault can be injected to gain access to the protected state *Final Round* (000) or the *Dangerous Don't-Care States* (101, 110, 111). AVFSM then performs static timing analysis (STA) to get the maximum path delay of each state FF and calculates SF for each transition. Table 3 shows the report generated by the AVFSM framework for three such VT. In the table, $Fault_{State}$ denotes the destination state for fault attack, and $Path_{Fs}(i)$ denotes the maximum delay path for the ith state FF.

Table 3: Reports of vulnerable transition, VT.

Transition	$Fault_{State}$	$PathViolated$	$PathOK$	$PathNoEffect$	SF
$T(010,001)$	000	$Path_{Fs}(0)$	$Path_{Fs}(1)$	$Path_{Fs}(2)$	0.12
$T(100,001)$	101	$Path_{Fs}(2)$	$Path_{Fs}(0)$	$Path_{Fs}(1)$	0.22
$T(100,001)$	000	$Path_{Fs}(0)$	$Path_{Fs}(2)$	$Path_{Fs}(1)$	-0.02

AVFSM then removes the transitions from VT set whose $SF < 0$ because fault injection attack is not feasible during these transitions in the implemented FSM (see subsection 4.2). After that AVFSM calculates VF_{FI} and VF_{Tro} for scheme 2.

For scheme 1, AVFSM analyzes all transitions in the extracted STG and reports that during the transitions *T(101,000), T(110,000)* and *T(111,000)* a fault can be injected to access the protected state *Final Round* (100). Because the parent state of the VT are don't-care states, AVFSM next searches for the transitions during which fault can be injected to cause transition to state $S_D = \{101, 110, 111\}$ and reports that there is no VT that can cause transition to these states. Table 4 summarizes our analysis for schemes 1 and 2.

It is clear from the above analysis that the scheme 1 implementation of the AES encryption module is more resilient to fault attack than scheme 2. Also, the existence of certain don't-care in scheme 2 makes it more vulnerable to Trojan attacks than scheme 1.

Table 4: Vulnerability analysis for scheme 1 and scheme 2 of AES.

	scheme 1	scheme 2
VF_{FI}	(0,0)	(58.9%,0.15)
VF_{Tro}	0	0.18

5.3 Case Study: II

Here we use AVFSM framework to perform the vulnerability analysis of a simplified FSM of an RSA encryption module implementing the Montgomery ladder algorithm as shown in [6]. This FSM is composed of 7 states, {*Idle, Init, Load1, Load2, Multiply, Square, Result*}. Here, the attacker's objective is to bypass the intermediate rounds of 'Square' and 'Multiply' states and access the 'Result' state to get either the key or premature result of RSA encryption. Therefore, for this FSM, the set of protected states is $P = \{Result\}$ and set of authorized states is $L = \{Square\}$.

We have used the following two encoding schemes {*000, 001, 010, 011, 100, 101, 110*} and {*001, 010, 011, 100, 101, 110, 000*} represented as scheme I and scheme II, respectively to implement the FSM. For brevity, we do not show the extracted state transition graph and detailed vulnerability analysis. The result reported by our AVFSM framework is shown in Table 5.

Table 5: Vulnerability analysis for scheme I and scheme II of RSA.

	VT	VF_{FI}	VF_{Tro}
scheme I	1	(10%,0.66)	0
scheme II	3	(30%,0.15)	0.1

It can be observed from Table 5 that scheme I has 1 VT with high SF while scheme II has 3 VT with comparatively low SF. In other words, the first scheme has only 1 VT but an attacker can more easily perform fault attack during this transition. On the other hand, the second scheme has 3 VT but it is relatively difficult to inject a fault during each transition. Therefore, both implementations are vulnerable to fault attacks.

6. LOW-COST MITIGATION APPROACH

In this section we propose a modification of the FSM that will mitigate fault injection and Trojan attacks. These attacks aim to access a protected state from an unauthorized state (protected states and authorized states have been defined in Section 2). In our proposed approach the state FFs are replaced by 'Programmable State FFs'. We define 'Programmable State FFs' as the state FFs which go to the Reset/Initial state if the protected state is tried to be accessed by any other state apart from the authorized states.

The operation of state FFs is as follows. During each state transition the next state logic bits appear at the input of the state FFs and at the positive or negative edge of the clock signal the present state bits (at the output of the state FFs) are updated to the next state logic values. The concept of 'Programmable State FFs' is based on decomposing the state FF into its Master-Slave latch configuration. During each half clock cycle the next state values are stored in the Master latches while the present state values are stored in the slave latches. During this time period, we can check if the next state is a protected state and if it is then verify whether the present state is the authorized state. If this condition holds, then the transition to the protected state is legitimate; otherwise it is not.

We propose an extension to the AVFSM framework which automatically replaces the state FFs with 'Programmable State FFs' to mitigate possible attacks against FSM. The work flow is as follows:

- **Step1:** AVFSM gets the set of protected state (P), authorized state (L) and the Reset/Initial state from the designer.
- **Step2:** AVFSM uses the following behavioral condition to check if the transition to protected state is legitimate or not.

$$sel = (next\_state == P)\&(present\_state != L)\ ?\ 1 :\ 0 \quad (8)$$

- **Step3:** AVFSM synthesizes the behavioral condition to get the circuit implementation of eqn. 8 .
- **Step4:** AVFSM places a MUX between each Master and Slave latch pair. One input of the MUX comes from the Master slave and other input is the logic bit of the Reset/Initial state. The select pin of the MUX is controlled by the circuit implementing eqn. 8.

The overall circuit of the 'Programmable State FFs' is shown in Fig 6(a). The 'blue' marked latches are placed to resolve the metastability problem. We simulated our proposed 'Programmable State FFs' in *Altera's Quartus* tool for the second encoding scheme for the FSM of AES encryption (see Section 5.2). For this FSM the

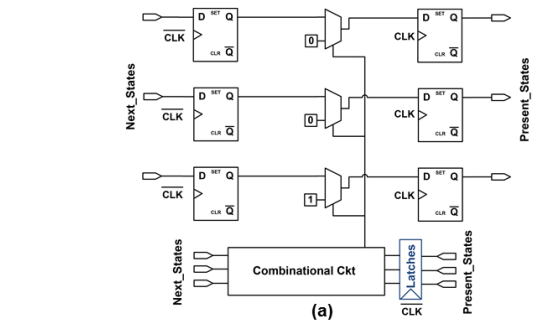

Figure 6: Proposed modification of the FSM to mitigate fault injection and Torjan attacks. (a) 'Programmable State FFs' circuit, (b) simulation result of the proposed solution.

protected state is 000, authorized state is 100 and the initial state (Wait Key) is 001. The simulation result is shown in Fig 6(b). Under normal operation the 'Programmable State FFs' performs identically as the traditional state FFs. At each positive edge of the clock signal the next state value is loaded to the present state. However, when the protected state (000) is tried to be accessed by an unauthorized state (101), the FSM goes to Wait Key state (001) instead of going to protected state (000) (see Fig 6(b)).

Note that 'Programmable State FFs' requires that the delay of the combinational circuit implementing eqn. 8 is less than half of the clock period. Also note that, this combinational circuit implements a relatively simple condition and therefore, its area and delay overhead is small compared to the whole design. The 'Programmable State FFs' structure itself is resilient to fault injection because the path between the Master and Slave latches is symmetric.

7. CONCLUSION

This is the first paper that systematically analyzes and evaluates vulnerabilities in the FSM against fault injection and Trojan attacks. Our proposed AVFSM framework allows the designer to find security vulnerabilities in the FSM at an early design stage. AVFSM also enables the designer to quantitatively compare the security of different implementations of the same design. If vulnerabilities exist in the design then our proposed mitigation technique can be applied to make the FSM secure against such attacks.

8. REFERENCES

[1] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems" in Lecture Notes in Computer Science, 1996.
[2] P. C. Kocher, et al., "Differential Power Analysis" in CRYPTO, 1999.
[3] D. Hely *et al.*, "Scan design and secure chip [secure IC testing]," in in Proc. 10th IEEE IOLTS, Jul. 2004.
[4] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," CRYPTO, 1997.
[5] R. Karri *et al.*, "Trustworthy Hardware: Identifying and Classifying Hardware Trojans", Computer, 2010.
[6] B. Sunar *et al.*, "Sequential circuit design for embedded cryptographic applications resilient to adversarial faults," IEEE Transactions on Computers, 2007.
[7] Z. Wang *et al.*, "Robust FSMs for cryptographic devices resilient to strong fault injection attacks," in On-Line Testing Symposium (IOLTS), 2010.
[8] C. Dunbar and G. Qu., "Designing Trusted Embedded Systems from Finite State Machines," in ACM Trans. Embed. Comput. Syst., 2014.
[9] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level," in Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013.
[10] T. Schneider and A. Moradi, "Leakage Assessment Methodology - a clear roadmap for side-channel evaluations," in CHES 2015.
[11] B. Yuce *et al.*, "TVVF: Estimating the vulnerability of hardware cryptosystems against timing violation attacks," in Hardware Oriented Security and Trust (HOST), 2015.
[12] *http://opencores.org/*.
[13] L. Yuan *et al.*, "An fsm reengineering approach to sequential circuit synthesis by state splitting," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2008.
[14] Y. Shi *et al.*, "A highly efficient method for extracting fsms from flattened gate-level netlist," in Circuits and Systems (ISCAS), 2010.
[15] L. Zussa *et al.*, "Investigation of timing constraints violation as a fault injection means", in DCIS 2012.