FSM Anomaly Detection using Formal Analysis

Farimah Farahmandi and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, USA

***Abstract*—Finite state machines (FSMs) control the functionality of the overall design. Any deviation from the specified FSM behavior can endanger the trustworthiness of the design. This is a critical concern when an FSM is responsible for controlling the usage or propagation of protected information (e.g. secret keys) in a secure component. FSM vulnerabilities can be created by a rogue designer or an attacker by inserting hardware Trojans in the FSM implementation. The vulnerability can also be introduced unintentionally by a CAD tool (e.g., when a synthesis tool is trying to optimize a gate-level netlist). In this paper, we present an efficient formal analysis framework based on symbolic algebra to find FSM vulnerabilities. The proposed method tries to find inconsistencies between the specification and FSM implementation through manipulation of respective polynomials. Security properties (such as a safe transition to a protected state) are derived using specification polynomials and verified against implementation polynomials. In a case of a failure, the vulnerability is reported. While existing methods can verify legal transitions, our approach tries to solve the important and non-trivial problem of detecting illegal accesses to the design states (e.g., protected states). We demonstrated the merit of our proposed method by detecting the vulnerabilities in various FSM designs, while state-of-the-approaches failed to identify the security flaws.**

I. Introduction

Integrated circuits (ICs) are deployed in a wide variety of systems designed for personal use as well as military and governmental purposes. To ensure security and privacy during computation and communication utilizing these systems, it is critical to assure the security of the ICs in these systems. However, ensuring the integrity of an IC is challenging due to the diversity of attacks and attack goals. Malicious modifications [1], side channel attacks such as power analysis [2] and timing analysis [3], debug infrastructure vulnerabilities [4] and fault injection attacks [5] can be exploited to affect security of a System-on-Chip (SoC), an Intellectual property (IP) or a microprocessor. A design can be resilient against such vulnerabilities when the security is considered from early design stages including controller and datapath design efforts.

Wide variety of solutions are proposed to protect datapath components [6], [7], [8], [9]. However, only a few studies addressed potential integrity issues of control circuits. Control circuits are required to be resilient against different types of attacks since they are responsible for controlling the functionality of the overall design and any deviation from the expected behavior can lead to severe impacts on security of the whole design. A Finite State Machine (FSM) of a secure design usually contains protected states which control proper handling of secret information. Fault injection attacks [5], existing EDA tools incompleteness [10] as well as designers' mistakes can compromise the security of a control circuit.

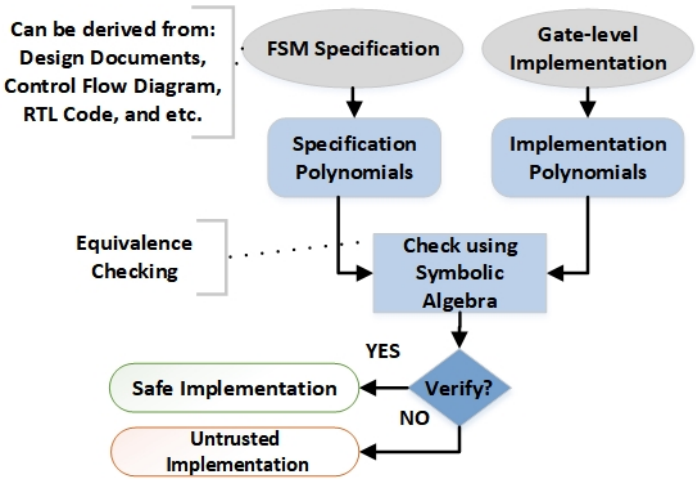

Fig. 1. Overview of FSM anomaly detection approach

An attacker's goal is to utilize existing FSM vulnerabilities to bypass authorized states and access the protected states illegally to weaken the security of the design or leak secret information such as cryptographic keys. Sunar et al. have shown that the secret key of RSA encryption algorithm [11] can be leaked when fault injection attack is used against the implementation of the Montgomery ladder algorithm [12]. It has been shown that some FSM encodings are more vulnerable toward fault injection attacks and an adversary can use the existing encoding vulnerabilities to have unauthorized access to the protected states [13]. Therefore, it is vital to identify and remove the vulnerabilities in the FSM architecture to protect them against any susceptibilities.

In this paper, we propose a formal approach to identify vulnerabilities in an FSM using symbolic algebra. Our proposed method models the specification of a given FSM as a set of polynomials ($\mathbb{F}_{spec}$) such that each polynomial is responsible for describing all of the valid states that can be reached. Each output of the FSM also can be represented using one specification polynomial. The specification polynomials can be derived from RTL codes as well as design documents. We also partition the gate-level implementation of an FSM based on the boundary of flip-flops, primary inputs, primary outputs and fanout-free regions. We model each region by a polynomial and add it to the set of implementation polynomials ($\mathbb{F}_{imp}$). In the next step, we use Gröbner basis theory [14] to check the equivalence between two sets $\mathbb{F}_{spec}$ and $\mathbb{F}_{imp}$. We reduce each specification polynomial F_{spec_i} using a set of implementation

polynomials. If the reduction leads to a non-zero remainder, there are some vulnerabilities in implementation of F_{spec_i}. Every assignment that makes the remainder non-zero reveal the conditions that can activate the hidden malfunction.

Our approach is fully automated and it is guaranteed to find hard-to-detect FSM vulnerabilities in the implementation of an FSM when existing equivalence checking approaches fail. Experimental results demonstrate the effectiveness of our approach. Figure 1 shows the overall flow of our approach which the anomaly detection can be formally performed using our proposed equivalence checking method.

The rest of the paper is organized as follows. We provide an overview of related work in Section II. Section III discusses about equivalence checking using symbolic algebra followed by description of threat model in Section IV. Section V illustrates our approach to detect FSM vulnerabilities. We show the effectiveness of our approach using the experimental results in Section VI . Finally, conclusion is provided in Section VII.

II. Related Work

There are limited efforts to identify and address the security vulnerabilities of a control circuit. Sunar et al. used Triple Module Redundancy (TMR) and parity checking methods to protect FSM of encryption algorithms against fault injection attacks [12]. However, the proposed technique introduces large area overhead (200%) and cannot detect other adversarial models such as hardware Trojans and vulnerabilities introduces by synthesis tools. In [15], a multilinear code selection algorithm is used to make cryptographic algorithm robust against fault injection attacks. However, this technique is not resilient against fault injection vulnerabilities caused by synthesis tools [10]. It has been shown that synthesis tools may insert additional *don't care* states in implementation of FSMs by using RTL don't care conditions and create assignments to optimize the gate-level netlist. At the same time, an adversary can use don't care states as a backdoor to access protected states and weaken the security of the overall design. In [10], authors use reachability as a trust metric to identify gate-level paths to protected states which do not exist in the RTL design. However, authors do not evaluate actual vulnerabilities caused by don't care states. They proposed an architectural change to state flip-flops in order to remove the access to the protected states from unprotected ones. Their proposed solution limits the functionality of the design. In [16], authors used mutation testing to detect existing hardware Trojans in unspecified functionality. However, mutation testing is very slow, and it may require significant manual intervention. Nahiyan et al. have proposed a state reachability analysis using ATPG tools [13]. They generate test patterns using the principle of n-detect-test [17] to extract the state transition graph (STG) of a given circuit. However, this option does provide any guarantees, e.g., in case one of their benchmarks they could not extract the whole STG. Sun et al. have proposed an FSM traversal technique using symbolic algebra [18]. However, their technique can only check the reachable states from a given state (e.g., initial state) and their technique cannot detect don't care states that may be introduced by synthesis tools. Similarly, they cannot detect hardware Trojans inserted in FSMs outputs. In this paper, we present a scalable formal approach that enables efficient FSM anomaly detection in state transition functions as well as FSM outputs.

III. Equivalence Checking using Gröbner Basis Reduction in Combinational Circuits

From the security point of view, it is extremely important to make sure that the design performs exactly the intended specification, nothing more nothing less. There are equivalence checking methods based on symbolic algebra that are successful to detect deviations from the specification for combinational circuits specially arithmetic circuits [19], [20], [21]. These methods map the equivalence checking problem to ideal membership testing and solve the problem using Gröbner Basis theory.

In order to check the equivalence between the implementation and specification of an arithmetic design using Gröbner Basis reduction, the specification and the implementation should be modeled as polynomials first. Specification polynomial F_{spec} represent the word-level (decimal) function of the arithmetic circuit where its variables are a combination of primary inputs and primary outputs. For instance, to represent the specification polynomial of a n-bit adder with primary inputs $A = \{a_0, a_1, ..., a_{n-1}\}$ and $B = \{b_0, b_1, ..., b_{n-1}\}$ and primary output $S = \{s_0, s_1, ...s_n\}$, $S = A + B$ can be used. Polynomial F_{spec} can also be written as $(2^n.s_n + ... + 2.s_1 + s_0) - ((2^{n-1}.a_{n-1} + ... + 2.a_1 + a_0) + (2^{n-1}.b_{n-1} + ... + 2.b_1 + b_0)) = 0$ where $\{a_i, b_i, s_i\}$ can be either zero or one (decimal values). Gate-level implementation of an arithmetic circuit can be modeled as a set of polynomials ($\mathbb{F}_{imp}$) by converting each gate to one polynomial as shown in Equation 1. Each variable x_is shows the gate input and each variable y_is shows the gate output where both input and output can get either zero or one decimal values and ${x_i}^2 = x_i$ (same for y_i).

$$\begin{aligned} &y_1 = \neg x_1 \rightarrow y_1 = (1 - x_1), \\ &y_2 = x_1 \wedge x_2 \rightarrow y_2 = x_1.x_2, \\ &y_3 = x_1 \vee x_2 \rightarrow y_3 = x_1 + x_2 - x_1.x_2, \\ &y_4 = x_1 \oplus x_2 \rightarrow y_4 = x_1 + x_2 - 2.x_1.x_2 \end{aligned} \tag{1}$$

To check the equivalence of f_{spec} and the gate-level implementation, f_{spec} is reduced over implementation polynomials ($\mathbb{F}_{imp}$) until it results in either zero polynomial or a non-zero polynomial (remainder) that contains only primary inputs is reached. The non-zero remainder shows that the arithmetic circuit implementation does not perform the exact specification. Therefore, the implementation is not trustworthy and there are some malfunctions in the gate-level implementation of the arithmetic circuit.

Example 1: Suppose that we want to verify the functionality of a full-adder implemented as shown in Figure 2. The specification polynomial can be formulated as: $f_{spec} := 2 * C_{out} + S - (A + B + C_{in}) = 0$ and implementation polynomials are computed based on Equation 1. The reduction procedure starts by substituting the most significant primary output using implementation polynomials output until a zero

remainder or a polynomial contains only primary inputs is achieved (procedure is shown in Equation 2). Note that reduction procedure follows the topological order of the circuit as: $\{Cout, S\} > \{W_3\} > \{W_2, W_1\} > \{A, B, C_{in}\}$. The zero remainder shows the success of the gate-level netlist in implementing the specification polynomial and trustworthiness of the implementation. ■

$$\begin{aligned} &step_0(f_{spec}) : 2.C_{out} + S - A - B - C_{in} \\ &step_1 : S - 2.w_3.w_2 + 2.w_3 + 2.w_2 - A - B - C_{in} \\ &step_2 : 2.w_2 + 2.w_2.A.B2.w_1.Cin + w_1 + 2.A.B - AB \\ &step_3(remiander) : 0 \end{aligned} \tag{2}$$

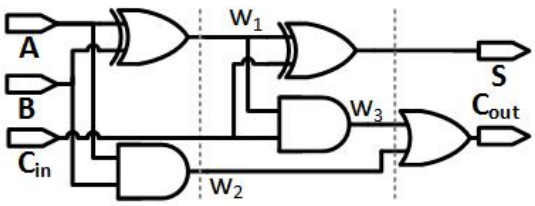

Fig. 2. Gate-level netlist of a full-adder

IV. Threat Model

In this section, we describe different categories of FSM vulnerabilities and show how an adversary can take advantage of these vulnerabilities to threaten the integrity of the overall design.

A state machine can be defined with six characteristics: an initial state S_{init}, set of possible states $\mathbb{S}$ where $S_{init} \in \mathbb{S}$, set of possible input events $\mathbb{I}$, a state transition function (F_T) that maps combination of states and inputs to states ($F_T : \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{S}$), a set of output events ($\mathbb{O}$) and an output function (F_O) that maps states and inputs to outputs ($F_O : \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{O}$). Based on the function F_T which defines transitions, each state S_i can be accessed through a set of immediate, authorized states as well as a set of specific input events. Set $\mathbb{A}_{S_i} = \{(S_j, I_j) | S_j \in \mathbb{S} \quad \& \quad I_j \in \mathbb{I}\}$ shows legal conditions to access state S_i and set $\mathbb{A}_{\mathbb{S}}$ shows all of the legal ways to access states $\mathbb{S}$. If state S_i can be accessed through the condition (S_m, I_m) where $(S_m, I_m) \notin \mathbb{A}_{S_i}$, it is a threat to the integrity of the design. In other words, state S_i should not be accessed through some illegal conditions/states which do not exist in the specification. From the security perspective, it is important that a design exactly performs as intended in the specification, nothing more nothing less. The extra access path to state S_i, (S_m, I_m) may endanger the integrity of the design as it may create a backdoor to access the critical secrets/assets. **In this paper, we consider illegal access paths as threat model, and our goal is to identify them using symbolic algebra.**

The illegal access ways may be introduced by synthesis tools [10]. Behavioral specification (e.g. RTL) of an FSM may contain don't care conditions where the assignment to the next state or the next expected output is not defined (we call such FSMs incomplete FSMs). A synthesis tool takes an incomplete FSM and tries to assign deterministic values to the don't care conditions and transitions to generate an optimized circuit. As a result, a synthesis tool may introduce extra states and transitions to the gate-level implementation of the FSM which do not exist in the behavioral specification. Formally, a synthesis tool may modify the set $\mathbb{A}_{\mathbb{S}}$ and convert it to $\mathbb{A}'_{\mathbb{S}}$. The extra set of access paths to the states of $\mathbb{S}$ can be computed as: $\mathbb{A}_M = \mathbb{A}'_{\mathbb{S}} - \mathbb{A}_{\mathbb{S}}$.

Set $\mathbb{A}_M$ (malicious access ways) can also be created/modified by a rogue designer or an attacker by inserting hardware Trojan in the FSM behavioral description as well as in the gate-level implementation of the FSM. The primary goal of the attacker is to create a backdoor to particular FSM states which may be triggered via an extremely rare input condition. The created backdoor may lead to a bypass of security protection of the design or create a denial of service. Moreover, malicious access ways can be set up by unintentional mistake of the designer. Example 2 illustrates potential threats in an FSM.

Example 2: The state transition diagram of a simple FSM is shown in Figure 3. The FSM has three states: G, C and protected state O representing with binary encoding 01, 10, and 00 respectively as shown in Figure 3. The FSM is responsible for checking a password before starting a specific operation. Operation state (O) should be accessed only from check password state (C) when a password is entered, and it is valid ($a = 1 \& b = 1$). An adversary may use the unspecified conditions to insert illegal transitions to gain access to the operation state (protected state) from the state G without even entering the correct password to bypass the security protection ($a = 1 \& b = 0$). On the other hand, the synthesis tool or the designer mistake can also introduce some unintentional illegal access ways (don't care states D) to the protected state and compromise the security of the design. With respect to the specification, $\mathbb{A}_O$ should be equal to: $\{(C, \text{“}a = 1 \& b = 1\text{”})$. However, there are illegal access ways to state O in FSM implementation which is equal to: $\mathbb{A}_{M_O} = \{(D, \text{“}a\text{”}), (G, \text{“}a = 1 \& b = 0\text{”})\}$. An adversary can compromise the security of the design by exploiting the existing vulnerabilities and attack the FSM. One of the possible attacks is fault injection attack [13]. The strategy is that the attacker tampers operating characteristics such as clock signal frequency, operating voltage or working temperature hoping to change different path delays and force the FSM to capture next state incorrectly. One example would be to force the FSM to go to the don't care states which have access to protected states or attack target states. For instance, an attacker can inject a fault during transition $01 \rightarrow 10$ ($G \rightarrow C$) to end up in don't care state 11 which has an immediate access to the protected state O and bypass password checking process in Example 2. The other possible attack is that the adversary inserts hardware Trojan by manipulating state transition graph in order to access certain states when a specific input event is triggered. In this case, the adversary is considered as an in-house rogue designer or an untrusted vendor/foundry. For instance, Example 2 shows that an adversary has inserted a Trojan that provides an illegal access way to state O from

state G. The Trojan is typically hard-to-activate (from the unspecified design space) with negligible effect on the design constraints such as area and power to avoid detection from existing verification and debug flow. ■

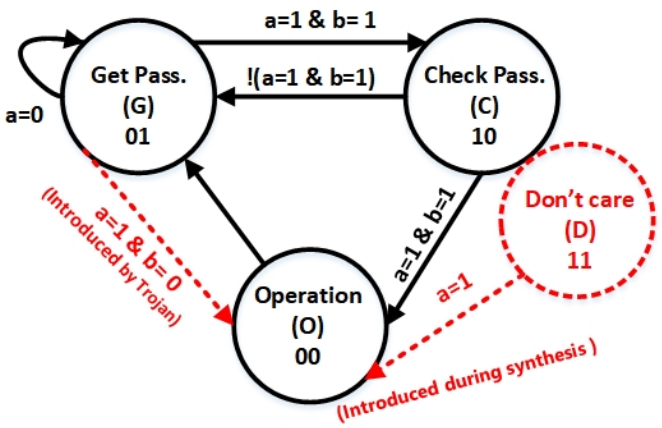

Fig. 3. The state diagram for checking a password in order to perform a specific operation. Potential vulnerabilities are shown with dotted lines.

Based on above observations, any deviation of FSM implementation from the specification (including extra access ways) can endanger the overall design integrity. In the rest of this paper, we propose a promising approach to analyze FSMs to find potential malicious functionality.

V. Finite State Machine anomaly detection

Although the presented approach of Section III is promising for verification of arithmetic circuit, applying it on a general sequential circuit is challenging due to several reasons. First, formulating the specification of a general circuit cannot be modeled as one simple and comprehensive polynomial. The specification may be modeled as a set of polynomials. However, finding the corresponding parts which are only responsible for implementing a special specification polynomial is not straightforward. Second, the implementation of a sequential circuit is not acyclic and it contains several loops which make the reduction operation infinite. Finally, time unrolling of the implementation is not efficient since it increases the design complexity and makes the equivalence checking inefficient. Moreover, existing Trojan may be activated after a large number of cycles (since the trigger condition is rare), therefore, there is no specific information about the required number of unrolling. In this paper, we try to address the above-mentioned challenges to apply symbolic algebra to verify the trustworthiness of any general FSM. We not only check the given FSM for the correct expected behavior, but we also analyze the FSM to find any potential malicious extra access ways that may endanger the security of the FSM (nothing more). Finding extra access path especially from don't care states cannot be found using any formal methods such as model checkers since they are not accessed through the normal operation path. To the best of our knowledge, this is the first attempt in utilizing symbolic algebra in finding vulnerabilities in FSMs. The remainder of this section describes the different parts of our approach: deriving specification polynomials, generating implementation polynomials and performing equivalence checking in order to ensure the correctness of implementation and finding potential extra vulnerabilities.

A. Deriving Specification Polynomials

The specification of an FSM can be extracted from its state transition diagram or from a high-level description of the design (e.g., HDL modules). State transition graph can be derived from the design documentation as well as other high-level behavioral description of FSM such as RTL codes. In other words, deriving specification polynomials does not require a golden design/netlist.

Modeling the whole FSM using only one specification polynomial is not possible without considering the time notation in the specification polynomial as transitions between different states may be dependent on binary values of a specific input variable over different clock cycles. For example, as it is shown in Figure 3, state C can be accessed from path $G \rightarrow C$ when in two consecutive clock cycles t_1 and $t_1 + 1$ such that $a = 0$ in t_1 and $a = 1, b = 1$ in $t_1 + 1$. Writing these conditions as a polynomial (part of the overall specification polynomial) without considering the timing will lead to a zero polynomial as $(1 - a).a.b = 0$. However, if we add timing notations to our variables, the implementation also has to be time unrolled to match with the specification which increases the complexity of the equivalence checking problem. As a result, representing the functionality of an FSM using one specification polynomial is not possible. We propose an approach to model the specification of the FSM using polynomials without time unrolling the design.

Transitions of an FSM can be decomposed as: $F_T = \bigcup_{i=1}^{n} \mathbb{A}_{S_i}$ where n is the number of states and $\mathbb{A}_{S_i}$ shows all of the possible access ways of state S_i and F_T is the transition function of the FSM. To derive a set of specification polynomials which represent the whole FSM, we model each of $\mathbb{A}_{S_i}$ as one polynomial representing the legal access ways to state S_i and we add it to the set $\mathbb{F}_{spec}$.

A valid transition to state S_i happens when the current state is one of the authorized states and the corresponding input conditions are valid. In other words, S_i will be reached in the next clock cycle when the current state is S_j and condition $C_{j \rightarrow i}$ where $(S_j, C_{j \rightarrow i}) \in \mathbb{A}_{S_i}$ are evaluated to true. Note that, we show the value of variable x in the next cycle using x' notation. Therefore, transition $S_j \rightarrow S_i$ is modeled to a polynomial as: $f_{S_j \rightarrow S_i} : S_i' - (S_j.C_{j \rightarrow i}) = 0$. The polynomial of each of the conditions exits in $\mathbb{A}_{S_i}$ should be XORed to each other to derive a polynomial representing the whole $\mathbb{A}_{S_i}$ since only one of them should be valid at the same time. We illustrate our approach using Example 3.

Example 3: In order to extract specification polynomials for FSM shown in Figure 3, we consider each of the states independently and write a polynomial to represent conditions which update the next value of the state. For example, state O should only be accessed from state C when $a = 1$ and

$b = 1$ or when the current state is state O and input a is equal to one. Since it should be accessed only from one of these conditions at a time, the conditions should be XORed to each other to show the effect of one condition at a time (the only exception is the condition of $a = 0$ in state G that will be ORed to other conditions since it works as the reset signal). The O' shows the next value of state O. The specification of the FSM shown in Figure 3 can be modeled as a set of three abstract polynomials ($\mathbb{F}_{spec} = \{f_G, f_C$ and $f_O\}$) as shown in Equation 3. ■

$$\begin{aligned}&\mathbb{F}_{spec} : \{f_G : G' - ((1-a) \vee (C.(1-a.b) \oplus O)) = \\ &G' - (1 - a + a.O + 2.a.b.C.O + a.C - 2.a.C.O - 1.a.b.C) = 0 \\ &f_C : C' - a.b.G = 0 \\ &f_O : O' - (a.b.C) = 0\}\end{aligned} \tag{3}$$

We will describe how specification polynomials are used to check security properties of an FSM in Section V-C. Before performing the equivalence checking, we need to refine specification polynomials to apply proposed FSM equivalence checking process since the proposed method requires that specification variables' names be the same as the corresponding variables in the implementation. We refine specification polynomials based on the FSM encoding style as well as corresponding names of state flip-flops in the implementation (name mapping between flip-flop names and corresponding variables in specification polynomials). We refine the variables which represent states in specification polynomials based on naming and encoding information that can be found in the high-level description of the design such as RTL modules as we describe in Example 4. As a result, the specification of FSM outputs can also be modeled with word-level specification polynomials based on state variables as well as primary inputs.

Example 4: Suppose that the RTL code shown in Listing 1 is the RTL version of the state machine shown in Figure 3. We can see that states G, C and O are encoded as $\{01, 10, 00\}$ respectively. The state variable and next states are presented using variables $\{s_0, s_1\}$ and $\{n_0, n_1\}$. Therefore, the variables shown in Equation 3 can be updated based on the above-mentioned information. For instance, variable G and next state variable G' can be modeled as $(1 - n_1).n_0$ and $(1 - s_1).s_0$, respectively. As a result, the specification polynomials shown in Equation 3 can be rewritten as shown in Equation 4. Note that, considering C encoded as $s_1.(1 - s_0)$ and O as $(1 - s_1).(1 - s_0)$, the term $-2.C.O$ as well as $2.a.b.O.C$ of F_G in Equation 3 are evaluated in updated specification polynomials). ■

$$\begin{aligned}&\mathbb{F}_{spec} : \{f_G : (1 - n_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) = 0 \\ &f_C : n_1.(1 - n_0) - (a.b.(1 - s_1).s_0) = 0, \\ &f_O : (1 - n_1).(1 - n_0) - (a.b.s_1.(1 - s_0)) = 0\}\end{aligned} \tag{4}$$

Specification polynomials can be extracted directly from the RTL modules by using some specific rules. The logical operations in *If* statements can be mapped to polynomials based on Equation 1. For example, by considering the encoding, line $G : if(a == 1'b1 \&\& b == 1'b1) n <= C$ can be modeled as equation $n_1.(1 - n_0) = a.b.(1 - s_1).s_0$ In the next step, the corresponding polynomials of *If Then Else* are XORed together to achieve the exclusive nature of these statements. The derived specification polynomials will be used in the equivalence checking procedure.

Listing 1. RTL module of FSM shown in Figure 3.

```
module fsm(input clock, a, b; output valid );
reg[1:0] s, n;
parameter O=2'b00, G=2'01, C=2'b10;
always @(a, b, s) begin
case(s)
  G: if(a == 1'b1 && b == 1'b1) begin
     n <= C;
  end else if (a == 0) begin
        n <= G; end
  C: if(a == 1'b1 && b=1'b1 )
        n <= O;
  else
     n <= G; end
  O: n <= G;
end
always @(posedge clock)
begin
  if(a==1'b0) s <= G;
  else s <= n; end
end
endmodule
```

B. Generation of Implementation Polynomials

Our goal is to partition the design and find the regions that are responsible for implementing each of the states and represent them as implementation polynomials. In order to perform this task, a mapping between state names and their corresponding gate-level state flip-flop names is needed. Here, we assume that the name of state inputs, outputs as well as state flip-flops are same between specification (RTL, state diagram, etc.) and implementation, or name mapping can be done based on existing methods in [22]. For the ease of the illustration, we explain how to extract the implementation polynomials when the FSM encoding is binary-encoding. Our proposed approach works for any state encoding.

After name mapping, we partition the gate-level implementation of the FSM based on state flip-flops. The state region construction starts from the input of the corresponding state flip-flop. The region construction continues with the inputs of the state flip-flop and moves backward recursively until it reaches to primary inputs or flip-flop outputs. The constructed region is converted to a polynomial by converting each of its gates to a polynomial as shown in Equation 1 and combining them to each other to create one polynomial representing the whole region. We illustrate our approach using Example 5.

Example 5: Figure 4 shows the gate-level netlist which implements the FSM shown in Figure 3. In the implementation, FSM states are encoded using binary scheme (two flip-flops are used to implement the functionality of three states shown in the state diagrams of Figure 3). The implementation is partitioned starting from the input of state flip-flop n_i and it is continued until reaching either primary inputs or outputs of state flip-flops (s_i). In the next step, the corresponding polynomial of each partition is derived by combining polynomials of each gate in the region to represent the functionality of

next state variables (n_i). The implementation polynomials are shown in Equation 5. ■

$$\mathbb{F}_{imp} : \{n_0 - (1 - a.b.s_1 - a.s_0 + a.b.s_0.s_1) = 0, \\ n_1 - (a.b.s_0 - a.b.s_0.s_1) = 0\} \tag{5}$$

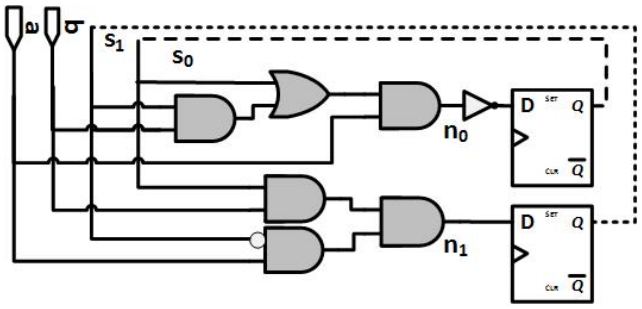

Fig. 4. Implementation of FSM in Figure 3 using binary encoding.

When a gate's output goes to more than one gate, it is called a fanout. A fanout-free region is a set of gates that are directly connected together. Therefore, we partition the implementation to fanout-free regions and model each of them as one polynomial. The corresponding polynomials of each next state variable (n_is) can be computed by combining the polynomials of the corresponding fanout-free regions. Polynomials of fanout-free regions are calculated in order to reduce the efforts of implementation polynomial generation since one fanout-free region may be used in constructing the functionality of several n_is. Note that, in the implementation shown in Figure 4, the functionality of each n_i is constructed with only one fanout-free cone.

Note that, the implemented functionality of FSM's outputs also can be formulated as a function of FSM inputs and states and presented as polynomials. In order to find implementation polynomials corresponding to FSM's outputs, each output gate is considered and traversed backward until it reaches to either input/output of state flip-flops or FSM inputs. The traversed gates are modeled using one polynomial showing the functionality of the corresponding output, and those polynomials are added to set $\mathbb{F}_{imp}$.

C. Equivalence Checking

From the security point of view, it is important to make sure that the implementation of a design performs exactly its specification. We check the functional equivalence between a control logic specification and its implementation in order to establish the trust of the control logic. In this paper, we formulate the FSM equivalence checking as ideal membership testing based on Gröbner Basis theory. Implementation polynomials $\mathbb{F}_{imp}$ are formed as an ideal I based on particular order $>$ (the topological order which exists in the implementation). FSM implementation is trustworthy if all of the specification polynomials in set $\mathbb{F}_{spec}$ are the member of ideal $I =< \mathbb{F}_{imp} >$.

In order to check the trustworthiness of the implementation, each specification polynomial F_{spec_i} from set $\mathbb{F}_{spec}$ is reduced over polynomials in $\mathbb{F}_{imp}$. All of the variables in specification polynomials (except primary inputs and flip-flops' outputs) are substituted with the corresponding functionality of the variable from the implementation polynomials. Note that, the reduction procedure is done using sequential polynomial division as shown in Section III. The reduction process continues until a zero remainder or a non-zero polynomial which contains a combination of primary inputs and flip-flop outputs is reached. If reduction F_{spec_i} over set $\mathbb{F}_{imp}$ results in a zero remainder, it means that F_{spec_i} belongs to the ideal $I =< \mathbb{F}_{imp} >$. In other words, set $\mathbb{F}_{imp}$ has successfully implemented the specification F_{spec_i}. Otherwise, the implementation of F_{spec_i} is not trustworthy (implementation is not equal to specification). If all of the remainders are equal to zero polynomials, it means that the overall implementation is equal to FSM's specification since set $\mathbb{F}_{spec}$ includes specification of the FSM states as well as specification of FSM's outputs (specification polynomials cover all specification space). Algorithm 1 shows the equivalence checking procedure.

Algorithm 1 FSM Equivalence Checking Algorithm

```
1: procedure EQUIVALENCE–CHECKING
2:     Input: Gate-level netlist imp and specification polynomials F_spec
3:     Output: FSM anomalies E
4:     F_imp=findImplementationPolynomials(imp)
5:     for each f_spec_i ∈ F_spec do
6:         r_i = reduction of f_spec_i over F_j s ∈ F_imp
7:         if (r_i! = 0) then
8:             T_i = findNonZeroAssignments(r_i)
9:             E.put(f_spec_i, T_i)
       return E
```

Algorithm 1 takes the gate-level netlist imp of a given FSM as well as the specification polynomials $\mathbb{F}_{spec}$ as inputs and tries to find any existing anomalies in the FSM. First, it computes the implementation polynomials ($\mathbb{F}_{imp}$) as described in Section V-B (line 4). In the next step, every specification polynomial f_{spec_i} (corresponding to state S_i) in $\mathbb{F}_{spec}$ is reduced over a set of implementation polynomials F_js using Gröbner Basis theory in order to find the remainder r_i (line 6). If the remainder is non-zero, it means that there are some malicious functionality in implementing specification polynomial f_{spec_i}. Every assignments that make the remainder non-zero, activates the malicious access path to S_i. The Algorithm stores the anomalies in the map $\mathbb{E}$ (lines 7-9).

Example 6: Consider the specification polynomials of Equation 4, gate-level netlist in Figure 4 as well as implementation polynomials shown in Equation 5. The Equation 6 shows the equivalence checking procedure with respect to topological order $\{n_1, n_0\} > \{s_1, s_0, a, b\}$. Note that, reducing of variables $\{n_1, n_0\}$ happen at the same time as their orders are the same. However, we show the reduction of F_{spec_1} in two steps

to illustrate the procedure better. ■

$$
\begin{aligned}
&F_{spec_1} : f_G : (1-n_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) \\
&stp_{11} : (1 - a.b.s_0 + a.b.s_0.s_1).n_0 - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) \\
&stp_{12} : (1 - a.s_0 - a.b.s_1 + a.b.s_0.s_1) - (1 - a.b.s_1 + a.b.s_0.s_1 - a.s_0) = 0 \\
&F_{spec_2} : f_C : n_1.(1-n_0) - (a.b.(1-s_1).s_0) \\
&stp_{21} : (-a.b.s_0.s_1 + a.b.s0) - (a.b.(1-s_1).s_0) = 0 \\
&F_{spec_3} : f_O : (1-n_1).(1-n_0) - (a.b.s_1.(1-s_0)) \\
&stp_{31} : (a.s_0 - a.b.s_0 + a.b.s_1) - (a.b.s_1 - a.b.s_1.s_0) = \\
&(remainder) : a.s_0 - a.b.s_0 + a.b.s_0.s_1
\end{aligned}
\tag{6}
$$

As shown in Equation 6, specification polynomials of states G and C are reduced to zero which means that they are safely implemented by the gate-level netlist. However, the reduction of specification polynomial of the protected state O results in a non-zero remainder. The remainder reveals potential vulnerabilities in the gate-level implementation of the design to access the protected state O. Every assignment that makes the remainder non-zero, discloses an unauthorized access path to the state O. Table I shows the malicious access paths. As it can be observed from Table I, don't care state $\{s_1, s_0\} = 2'b11$ can access the protected state O due to synthesis tool optimization (when input a is true). There is another malicious access path to the state O from state G when $a = 1$ and $b = 0$. This extra access is a hardware Trojan that was inserted by an adversary or a rogue designer.

TABLE I
MALICIOUS ACCESS PATHS TO THE PROTECTED STATE O SHOWN IN FIGURE 3

s_1	s_0	a	b
1	1	1	X
0	1	1	0

VI. EXPERIMENTS

A. Experimental Setup

In order to evaluate the effectiveness of our FSM anomaly detection approach, we have implemented the proposed Algorithms using Java. Our experiments were run on a PC with Intel core i7 and 16 GB memory. We have applied our method on various FSM benchmarks from *OpenCores* [23]. The benchmarks are described using RTL modules (that we treat as the specification). To obtain the gate-level implementation, we synthesize RTL modules using *Synopsys Design Compiler* [24]. We extract specification polynomials from RTL modules of FSM benchmarks considering their state transitions and output assignments. We have implemented a Java program such that we define the valid transitions to states in the form of abstracted polynomials and it generates one specification polynomial representing all of the logical transitions to a given state. The same approach was used to produce the specification polynomials for FSM outputs. On the other hand, implementation polynomials are driven automatically from the synthesized gate-level netlist using our proposed framework. In order to generate implementation polynomials, gate-level netlist is partitioned into the fanout-free regions which are restricted to flip-flops boundaries as well as primary input and primary outputs. We use fanout-free regions to reduce the number of implementation polynomials. We reduce specification polynomials over a set of implementation polynomials and each non-zero remainder represents an FSM security threat. The goal is to find the assignments to activate the vulnerabilities (if any).

B. Results

We have conducted two sets of experiments based on whether the vulnerability is introduced by the synthesis tool (unintentional) or an attacker (intentional). In the first set of experiments, the gate-level implementations are Trojan-free, and all the potential vulnerabilities are caused by the synthesis tool. Note that different encoding styles and values can create different vulnerabilities. In the second set of experiments, we have inserted hardware Trojans in state transitions as well as state outputs of the implementations in order to show the effectiveness of our approach. The results are shown in Table II and Figure 5, respectively.

Table II represents the result of proposed FSM equivalence checking approach for eight different benchmarks. The first column shows the type of the benchmark. The second column represents the encoding style of the FSM design. We have considered binary and one-hot encoding methods to show that our proposed approach is not dependent on the encoding approach. The third, fourth and fifth columns represent the number of gates, number of state flip-flops, and the number of states, respectively. The sixth column represents the number of transitions in the FSM design. The next two columns indicate the number of don't care states and don't care transitions that our method finds, respectively. Note that our method does not report the don't care states that are not connected to any other states. Finally, the last column shows the CPU time that our proposed equivalence checking (EQ) approach to find anomalies in FSM benchmarks.

To show that our proposed approach can also detect hardware Trojans inserted in the state transition function as well as in the logic that generates the outputs of the FSM, we inserted hardware Trojans by exploiting the unspecified functionality of different benchmarks. Figure 5 shows the required time to detect the injected Trojan. The attributes of the benchmarks are the same as shown in the Table II.

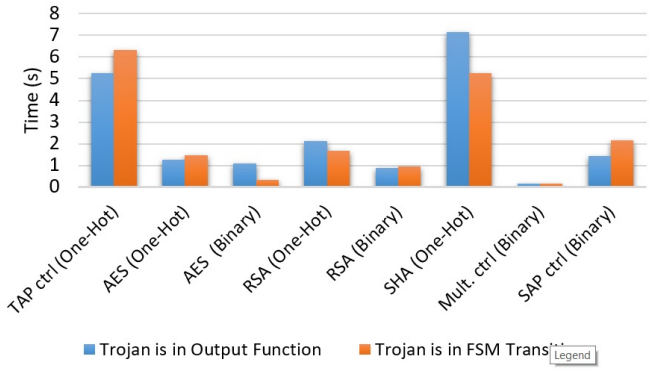

Fig. 5. Time required to detect hardware Trojans in output logic and state transition function.

TABLE II
RESULT OF THE PROPOSED FSM ANOMALY DETECTION TECHNIQUE USING EQUIVALENCE CHECKING.

Benchmark	Encoding	#Gates	#FF	#Sts	#Trans.	Our Approach		
						DC Sts	DC Tran.	EQ (s)
TAP controller	One-Hot	136	16	16	33	3	6	80.63
AES Encryption	One-Hot	88	5	5	11	0	0	6.26
AES Encryption	Binary	60	3	5	11	3	6	5.03
RSA Encryption	One-Hot	114	7	7	9	0	0	18.48
RSA Encryption	Binary	76	3	7	9	1	1	6.2
SHA Digest	One-Hot	153	7	7	47	121	121	50.89
multiplier Controller	binary	52	3	5	8	3	3	1.85
SAP controller	Binary	135	4	12	25	0	0	17.23

The experimental results demonstrated that our approach could detect the hidden vulnerabilities introduced by synthesis tool optimization while Formality fails to detect them. Note that some state encodings are more likely to have vulnerabilities caused by synthesis tools. For example, the synthesis tools tend to map all of the don't care states to a state with all zero's encoding (e.g. 3'b000) assuming that the state represents reset or ideal state. If the protected state is mapped using this encoding, there may be a direct access to the protected state from some don't care state caused by the synthesis tool.

VII. CONCLUSION

It is critical to make sure that FSMs are correctly implemented, and there is no deviation from the specified functionality of the FSM since any unexpected functionality can endanger the integrity of the whole design. FSM vulnerabilities can be caused intentionally through an adversary by inserting hardware Trojan in the implementation or unintentionally using CAD tools such as synthesis tools. In this paper, we presented an approach to formally detect anomalies in finite state machines using symbolic algebra. Our proposed approach models the specification of an FSM as a set of polynomials such that each polynomial represents all of the valid transitions to one of the states of the FSM. We modeled the implementation of an FSM as a set of polynomials. We check the equivalence of the specification polynomials and implementation polynomials using Gröbner basis theory. We have showed our approach can detect hidden vulnerabilities created by both synthesis tools or an adversary.

VIII. ACKNOWLEDGMENTS

This work was partially supported by grants from National Science Foundation (CNS-1441667), Semiconductor Research Corporation (2014-TS-2554) and Cisco.

REFERENCES

[1] R. Karri, J. Rajendran, K. Roseland, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," in *IEEE Computer*, 2010, pp. 39–46.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[3] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[4] J. Backer, D. Hély, and R. Karri, "Secure design-for-debug for systems-on-chip," in *IEEE International Test Conference (ITC)*. IEEE, 2015, pp. 1–8.

[5] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual International Cryptology Conference*. Springer, 1997, pp. 513–525.

[6] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 112.

[7] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016.

[8] F. Farahmandi, Y. Huang, and P. Mishra, "Trojan localization using symbolic algebra," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 591–597.

[9] Y. Huang, S. Bhunia, and P. Mishra, "Mers: statistical test generation for side-channel analysis based trojan detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 130–141.

[10] C. Dunbar and G. Qu, "Designing trusted embedded systems from finite state machines," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, p. 153, 2014.

[11] E. Brickell, "A survey of hardware implementations of rsa," in *Advances in CryptologyCRYPTO89 Proceedings*. Springer, 1990, pp. 368–370.

[12] B. Sunar, G. Gaubatz, and E. Savas, "Sequential circuit design for embedded cryptographic applications resilient to adversarial faults," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 126–138, 2008.

[13] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "Avfsm: a framework for identifying and mitigating vulnerabilities in fsms," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, p. 89.

[14] D. Cox, J. Little, and D. O'shea, *Ideals, varieties, and algorithms*. Springer, 1992, vol. 3.

[15] Z. Wang and M. Karpovsky, "Robust fsms for cryptographic devices resilient to strong fault injection attacks," in *2010 IEEE 16th International On-Line Testing Symposium*. IEEE, 2010, pp. 240–245.

[16] N. Fern and K.-T. T. Cheng, "Detecting hardware trojans in unspecified functionality using mutation testing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 560–566.

[17] S. C. Ma, P. Franco, and E. J. McCluskey, "An experimental chip to evaluate test techniques experiment results," in *Test Conference, 1995. Proceedings., International*. IEEE, 1995, pp. 663–672.

[18] X. Sun, P. Kalla, and F. Enescu, "Word-level traversal of finite state machines using algebraic geometry," in *High Level Design Validation and Test Workshop (HLDVT), 2016 IEEE International*. IEEE, 2016, pp. 142–149.

[19] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 145.

[20] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using grobner bases," 2016.

[21] F. Farahmandi and B. Alizadeh, "Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 83–96, 2015.

[22] T. Meade, S. Zhang, and Y. Jin, "Netlist reverse engineering for high-level functionality reconstruction," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 655–660.

[23] *OpenCores*, http://opencores.org.

[24] https://www.synopsys.com/support/training/rtl-synthesis/design-compiler.html.