ARC-FSM-G: Automatic Security Rule Checking for Finite State Machine at the Netlist Abstraction

Rasheed Kibria, Farimah Farahmandi, and Mark Tehranipoor
Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida
Email: rasheed.kibria@ufl.edu, farimah@ece.ufl.edu, tehranipoor@ece.ufl.edu

***Abstract*—Modern system-on-chip (SoC) designs are becoming prone to numerous security threats due to their critical applications and ever-growing complexity and size. Therefore, the early stage of the design flow requires comprehensive security verification. The control flow of an SoC, generally implemented using finite state machines (FSMs), is not an exception to this requirement. Any deviations from the desired flow of FSMs can cause serious security issues. On the other hand, the control FSMs may be prone to fault-injection and denial-of-service (DoS) attacks or have inherent information leakage and access control issues at the gate-level netlist abstraction. Therefore, defining a set of security rules (guidelines) for obtaining FSM implementations free from particular security vulnerabilities after performing logic synthesis is crucial. Unfortunately, as of today, no solution exists in the state-of-the-art domain to verify the security of control FSMs. In this paper, we propose a set of such security rules for control FSM design and a verification framework called ARC-FSM-G to check for those security rule violations at pre-silicon to prevent any security vulnerabilities of FSM against fault-injection, access control, and information leakage threats. Experimental results on several benchmarks varying in size and complexity illustrate that ARC-FSM-G can effectively check for violations of all the proposed rules within a few seconds.**

***Index Terms*—Security Rules, Finite State Machine, Gate-Level Netlist Analysis, Security Validation**

I. Introduction

System-on-Chips (SoCs) are integral parts of state-of-the-art *Internet of Things (IoT)* devices and their applications. Aside from the security requirements of these smart devices and their applications, an SoC might possess tens of intellectual property (IP) cores. The IP cores (analog, memory, digital, re-configurable fabric, etc.) may have distinct and complex functionalities, may contain security-critical information, and can introduce unique security concerns. Furthermore, vulnerabilities might get introduced when such IP cores interact with each other under real-world workloads. Therefore, the security of SoCs must be validated at the pre-silicon design phase before manufacturing and deployment in real-world systems. Additionally, implementing a secure SoC is challenging since vulnerabilities might appear at various stages of its design lifecycle. Generally, SoC vulnerabilities can be classified into multiple categories: information leakage, access control violations, side-channel leakage, hidden malicious functionalities, test and debug structures exploitation, and susceptibility to fault-injection attacks [1]–[3]. Some of these security flaws may be introduced accidentally by the mistakes of the designers or a lack of awareness of potential security issues. Moreover, computer-aided design (CAD) tools may inadvertently introduce new vulnerabilities in an SoC [1].

Traditionally, the very large-scale integration (VLSI) design industry devotes significant time and effort to SoC verification to fulfill several requirements, such as functional correctness, power, performance, and area (PPA) validation. With the continuous shrinking of device geometry, the size of an SoC is increasing as numerous complex and large-scale IPs are being integrated. Consequently, it is challenging for the verification engineers to perform the functional verification of the entire SoC once. On the other hand, verification has become much more challenging for assuring security [4]. Due to the diversity of attacks and lack of rules and metrics, manual approaches are mostly used to detect such issues. However, manual detection of security bugs is challenging, ineffective, and not scalable. Furthermore, the verification engineers are typically unaware of potential security concerns at the early stage of the design process, such as register-transfer level (RTL) designs and in the post-synthesis gate-level netlists. It is crucial to fix potential security issues as soon as possible in the VLSI design flow since detection and fixing of security bugs in the later stages generally come at a higher cost and effort, which can be thought of as increasing by a factor of ten after each phase, as a rule of thumb [5]. Verifying an SoC's control logic is one of the most crucial tasks in security verification since the security of the entire SoC can be compromised if its control logic can be successfully attacked. The control logic of an SoC, realized using finite state machines (FSMs), is susceptible to fault-injection and denial-of-service (DoS) attacks or may possess inherent information leakage or access control issues [9], [18].

The design of an FSM can be functionally correct at the RTL abstraction. Still, the state-of-the-art commercial synthesis tools might introduce particular security-critical bugs at the gate-level netlist, which may cause the implementation to be vulnerable to fault-injection and DoS attacks or suffer from access control and information leakage problems. Hence, after performing logic synthesis, a security verification framework for the control logic of an SoC is required to detect and enable fixing such security bugs. These bugs can be fixed by modifying the employed state encoding scheme or tweaking the FSM design at the RTL abstraction. Unfortunately, such a promising solution is absent in the state-of-the-art SoC security verification domain. Moreover, a set of security rules (guidelines) must be defined to fix those security bugs and assist in achieving secure implementation of control FSMs of an SoC. In this paper, we propose a number of such security rules and a verification framework called **A**utomatic Security **R**ule **C**hecker for **F**inite **S**tate **M**achine Design at the **G**ate-level Netlist Abstraction (ARC-FSM-G) to effectively and quickly detect violations of the proposed rules. Specifically, our significant contributions in this paper are as follows-

- Identification of the origin of the security-critical bugs present in the gate-level netlist implementation of FSMs;
- Developing a set of rules to obtain a secure implementation of FSMs free from security-critical bugs;
- Developing an automated framework named ARC-FSM-

G to validate the proposed rules after logic synthesis;
- Demonstrating the efficacy of ARC-FSM-G on 15 open-source designs from [23]–[25] and other resources varying in size, complexity, and functionality.

The rest of the paper is organized as follows. Section II presents definitions of the terminologies used in this paper. In Section III, we discuss the underlying motivation behind this work. Section IV presents the proposed security rules for control FSM design to be followed at the gate-level netlist abstraction. Section V illustrates our proposed ARC-FSM-G framework. We present experimental results with algorithmic complexity and efficacy analysis of ARC-FSM-G in Section VI. Finally, Section VII concludes the paper.

II. Definitions and Preliminaries

Finite State Machine (FSM): From analytical viewpoint, a *Finite State Machine* is a 6-tuple entity (S, I, O, s_0, ϕ, λ). Here, S represents a finite set of states, I stands for a finite set of inputs, O denotes a finite set of outputs, s_0 signifies the reset state of the FSM, $\phi : S \times I \rightarrow S$ is the function that determines the state transition conditions between the states of the FSM, and λ denotes the function determining the output logic. The generic architecture of an FSM is shown in Fig. 1.

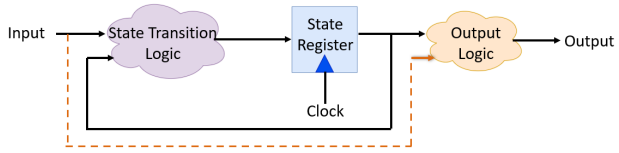

Fig. 1: General architecture of an FSM. The orange dashed line exists only in the generic architecture of the Mealy FSM.

State Transition Graph (STG): From the mathematical perspective, the *State Transition Graph (STG)* of an FSM is a directed graph where each vertex stands for a particular state $s \in S$, and each edge presents a specific state transition, $t = T(s_i, s_j)$ from the present state s_i to the next state s_j [1]. The STG of a particular control FSM is illustrated in Fig. 2.

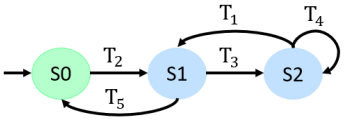

Fig. 2: State transition graph (STG) of a certain control FSM.

Don't-Care State: The states of a control FSM specified in the RTL description are the designer-defined FSM states. However, don't-care states can be added if the FSM is incomplete. A particular control FSM is defined as incomplete if the number of states present in the RTL description of the FSM is not equal to the maximum allowable one according to the size of the FSM state register. An incomplete FSM's unused (unspecified) states are termed don't-care states.

Protected State: A *Protected State* of a particular control FSM is a certain state considered crucial to be protected from a security viewpoint. Such a state can be where a security-critical signal is asserted, or bypassing or illegally accessing it may compromise the overall security of the design. The designer should identify the security-critical states of the control FSM to define the protected states of the FSM based on such intuition. A state of the control FSM, which should have access to the protected state, is an authorized state. For example, for the control FSM of SHA-512 design [24], the state of the FSM, which indicates that the encryption operation is completed, can be regarded as the protected state. It is because such an indication is vital from the security viewpoint. The encryption operation can be bypassed by accessing this state from an unspecified (don't-care) state without going through the required sequence of states [1]. A designer can define single or multiple protected states based on the functionality of the control FSM. The other states of the FSM defined in the RTL design will be treated as unprotected (or normal) states.

Security Rules: In this paper, *Security Rules* are defined as a set of guidelines to make the design of an FSM free from certain security-critical bugs, which are sometimes introduced at the netlist implementation after logic synthesis. Such security rules are validated at the netlist abstraction, and violations of these rules indicate that such security bugs exist in the design, which may potentially cause the final implementation of the FSM to be prone to various known security threats. These security rules focus on particular threat models and make the designer aware of security issues in the FSM implementation. Moreover, such rules will aid the designers in minimizing the possibility of obtaining an insecure FSM implementation by adopting appropriate countermeasures during RTL design.

III. Previous Work and Motivation

In the state-of-the-art SoC security verification domain, numerous security verification techniques have been proposed to identify security-critical bugs. These techniques can be broadly classified into 3 categories: property-based formal verification [7]–[9], information flow tracking (IFT) [10], [11], and run-time or dynamic detection [12]–[14]. These proposed techniques heavily rely on simulation or are suitable for in-field operations. First, the property-based formal security verification methodology requires the engineers to understand the design-under-verification (DUV) quite well. The verification engineers must write security properties to check whether the security requirements are met for specific threat models [7]–[9]. The quality of the security properties depends on the knowledge and experience of the verification engineers [9]. Secondly, the IFT techniques [10], [11] increase the complexity of the design as all the input variables need to be tainted for IFT utilization. Therefore, the IFT techniques are not scalable due to the high design overheads (timing, power, and area) required by such instrumentation and associated with higher computational time and memory resource utilization. Finally, the fuzz testing and run-time detection methods [12]–[14] inject invalid or unexpected inputs into an SoC to reveal vulnerabilities and require simulation to detect security bugs. These techniques are unsuitable for identifying particular security-critical bugs of control FSMs at the gate-level netlist. These bugs can exist not only in the RTL codes but also in the gate-level netlist implementation.

However, the static analysis-based methodology can provide a promising solution to detect specific security bugs in such a scenario if performed by analyzing the RTL source codes and the FSM STGs obtained after logic synthesis. Static analysis is a powerful, very fast, and well-established technique in the software domain. A lint tool based on static analysis for analyzing source codes written in *C* was first proposed in 1978 [15]. Since then, the linting concept has been adopted in several languages to analyze source code to detect specific bugs rapidly. The SoC verification domain was not an exception to adopting lint technology since linting helps to reduce

TABLE I: The proposed eight security rules for control FSM design to be validated after performing logic synthesis.

Source	Class	ID	Description
State encoding issues	I	R1	When state transition occurs between two consecutive unprotected states, the Hamming Distance (HD) between them should be '1'
		R2	When state transition occurs between two consecutive unprotected states, the Fault-Injection Feasibility (FIF) metric should be '0'
Accessibility issue	II	R3	A protected state should not be accessed by any unauthorized state
Bypassing issue	III	R4	The sequence of states specified by the designer should be maintained properly
Structural issues in STG	IV	R5	Dead (inactive) states should be absent in the extracted state transition graph (STG) of a control FSM from the netlist
		R6	Unreachable states with a transition to a non-reset state should not exist in the obtained STG of a control FSM from the netlist
		R7	States with static deadlock conditions should not be present in the extracted STG of a control FSM from the netlist
		R8	Group of states with a dynamic deadlock loop that includes a protected state should be absent in the gate-level STG of an FSM

overall verification time and effort by fixing certain functional bugs at the very early design stage. Note that some functional bugs can only be detected using static analysis. State-of-the-art commercial lint tools, such as *SpyGlass* from *Synopsys* [6], are excellent examples that incorporate static analysis techniques with formal methodology.

Such tools perform functional and structural checks to detect bugs. These tools focus on fixing functional bugs and can detect custom or pre-defined rule violations focusing primarily on logic synthesis issues. Nevertheless, such tools are unaware of security issues and cannot detect security bugs introduced in the FSM design by CAD tools after logic synthesis. These bugs may arise from state encoding issues or hidden don't-care states and transitions, as presented in Section IV, which may eventually make the design prone to fault-injection or DoS attacks or cause the design to have inherent information leakage and access control issues [1], [9], [18]. It motivated us to identify such security bugs and develop a static analysis-based framework named ARC-FSM-G to detect those without requiring simulation in the analysis process. After performing logic synthesis, we assume the designer will use this proposed security verification framework for control FSM design. Therefore, the design netlist and its associated RTL codes are accessible to the designer. Identifying security issues for the datapath logic of an SoC is beyond the scope of this paper.

IV. Security Rules for Control FSM Design

Security rules for control FSM design can act as guidelines for designers who do not have security expertise. As mentioned in Section II, we propose these security rules to be validated at the gate-level netlist abstraction and make the designer aware of particular security-critical bugs existing in the FSM implementation obtained after the logic synthesis. We have categorized the proposed security rules into four classes and provided them identifiers (IDs) from R1 to R8, as presented in Table I. Such security-critical bugs may arise from state encoding issues (see R1 and R2) or hidden don't-care states and transitions introduced after logic synthesis by state-of-the-art CAD tools (see R3 to R8) and can be fixed by adopting proper countermeasures in the RTL design stage. Furthermore, such bugs do not require simulation and test vectors to get detected. These rules will help the designers to achieve a secure implementation of FSMs resilient against certain fault-injection and denial-of-service (DoS) attacks and free from potential information leakage and access control issues. The proposed security rules are discussed as follows.

R1: *When state transition occurs between two consecutive unprotected states, the Hamming Distance (HD) between them should be '1'*. A control FSM can be prone to fault-injection attacks due to state encoding issues [1], [9], [18]. This rule focuses on making the FSM design resilient against setup time violation-based fault-injection attack [17]. It requires low-cost equipment to perform such an attack and poses a severe security threat. If the *Hamming Distance (HD)* between two

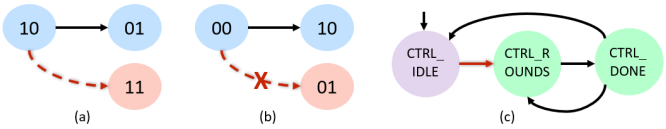

Fig. 3: An attacker may perform setup time violation-based fault-injection attack if the HD between two unprotected states of an FSM is greater than 1. In (a), such an attack is possible as HD is 2, but in (b), it is impossible since HD is 1. The control FSM of SHA-512 from [24] is shown in (c), which might be susceptible to this attack.

unprotected states is greater than 1, then the control FSM will be susceptible to this type of fault-injection attack. It has been illustrated in Fig. 3. An attacker can cause a setup time constraint violation for the most significant bit (MSB) state flip-flop while maintaining the setup time constraint for the least significant bit (LSB) state flip-flop to inject such a fault successfully [1]. In Fig. 3(a), the current and next state's LSB gets flipped. Hence, an adversary can perform the setup time violation-based attack at the LSB position to drive the FSM to the '11' state, which might be unwanted and lead to a potential security breach. Since the LSB remains unchanged, such an attack is not possible for Fig. 3(b). It can be mathematically modeled as finding and checking the HD between two unprotected (normal) states in a particular state transition, thus leading to this security rule. Hence, violations of this rule will warn the designers of the presence of this security bug. It can be fixed by utilizing the secure state encoding scheme proposed in [18]. For example, as shown in Fig. 3(c), the unprotected-to-unprotected state transition marked as red indicates a violation of R1, if the states of the control FSM of SHA-512 design from [24] are encoded as $\{$*CTRL_IDLE, CTRL_ROUNDS, CTRL_DONE*$\}$ = $\{$*01, 10, 00*$\}$. Those states can be encoded as $\{$*CTRL_IDLE, CTRL_ROUNDS, CTRL_DONE*$\}$ = $\{$*00, 01, 11*$\}$ to satisfy R1 when 'CTRL_DONE' is regarded as the protected state.

$$FIF = \prod_{i=0}^{n-1} [(S_{xi} \odot S_{pi}) + (S_{yi} \odot S_{pi})] \quad (1)$$

R2: *When state transition occurs between two consecutive unprotected states, the Fault-Injection Feasibility (FIF) metric should be '0'*. The FIF metric aims to identify the potentially vulnerable state transitions in which a fault can be injected to access a protected state. Suppose the *FIF* metric is 1 for a particular unprotected-to-unprotected state transition. During such a transition, an attacker may perform the mentioned setup time violation-based fault-injection attack (see R1) to access a protected state illegally utilizing the FSM don't-care states [1], [18], [22]. As proposed in [1], the *Fault-Injection Feasibility (FIF)* metric is defined according to Eq. 1, where n, S_x, S_y, and S_p denote the state register width, the state encoding bits of the source state, destination state, and protected state

respectively. For calculating the *FIF* metric, bit-wise XNOR of the encoding values of a protected state and the source state of a particular unprotected-to-unprotected state transition is calculated first. Next, the bit-wise XNOR of the encoding values of the protected state and the destination state of that transition is calculated. Then, the bit-wise OR value of these parts is obtained, and the logical AND of all the bits present in the result is calculated. Like R1, this rule targets to make the FSM design robust against setup time violation-based fault-injection attacks. However, R1 and R2 must be satisfied individually due to the computational methods' differences.

However, Eq. 1 applies only when a single protected state is defined. We have extended this metric to support scenarios if the designer specifies multiple protected states. Analytically, it can be done by calculating the final *FIF* metric value by taking the logical OR of all the individual *FIF* metric values arising from considering a single protected state once at a time. A warning will be provided to the designers in case of a violation of R2 that the control FSM implementation might be vulnerable to the setup time violation-based fault-injection attacks. The designer can clear this security bug by adopting the secure state encoding scheme proposed in [18]. For instance, the red-colored unprotected-to-unprotected state transition shown in Fig. 3(c) presents a violation of R2, if the states of the control FSM of SHA-512 design from [24] are encoded as $\{$*CTRL_IDLE, CTRL_ROUNDS, CTRL_DONE*$\}$ = $\{$*01, 10, 00*$\}$. Such states can be encoded as $\{$*CTRL_IDLE, CTRL_ROUNDS, CTRL_DONE*$\}$ = $\{$*00, 10, 11*$\}$ to satisfy the rule violation for R2 when 'CTRL_DONE' is treated as the only protected state. Last but not least, it needs to be noted that the number of options for secure state encoding of an FSM can be quite limited in certain scenarios. In those cases, using the secure state encoding approach proposed in [18] to satisfy the R1 and R2 rules will increase the overall area of the design. Nonetheless, the area overhead after using this secure encoding approach is less than 2% with no delay overhead and a negligible effect on overall power consumption.

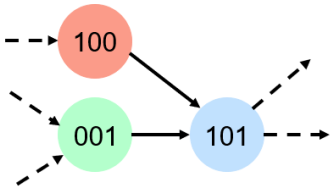

Fig. 4: An unauthorized state accesses a protected state of a security-critical FSM. The state '101' is a protected state with a single authorized state '001'. However, the '101' state is being accessed illegally by the unauthorized state '100', which may lead to information leakage and access control issues.

R3: *A protected state should not be accessed by any unauthorized state*. As mentioned in Section II, a protected state of an FSM, defined by the designer, is a critical state of a control FSM, which is wholly or partly responsible for controlling the critical operation of a particular design. The designers should also identify and specify the authorized states of an FSM as required to check for this security rule violations. The rest of the states of the FSM are treated as unauthorized states to have access to a particular protected state. As shown in Fig. 4, the control FSM is vulnerable to potential information leakage and access control issues since the protected state '101' is accessible by the unauthorized state '100'. Such a scenario will violate this security rule R3. If an unauthorized state can access a particular protected state in the gate-level netlist abstraction, it is dangerous from the security viewpoint. The reason is that an attacker may implant a Trojan to access that unauthorized state and may have access to the protected state, bypassing the typical execution sequence of the FSM. It may eventually lead to information leakage and access control issues. Furthermore, during logic synthesis and optimization, sometimes CAD tools can introduce don't-care states and transitions, potentially adding vulnerability to the FSM implementation. It is because a protected state can be accessed illegally through the don't-care states [1], [18], [22]. Hence, a violation of this security rule implies the presence of a security-critical bug in the FSM implementation. This security rule can be satisfied by reverting to the reset state of the FSM in case of violations.

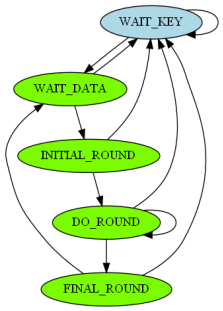

Fig. 5: The STG of the AES encryption operation controller FSM from [25] extracted by *RTL-FSMx* [19].

R4: *The sequence of states specified by the designer should be maintained properly*. It is crucial from the security viewpoint since improperly implementing standard security compliance may lead to information leakage issues [1], [18], [22]. For example, for the IP core obtained from [25] to implement the AES encryption operation, the control FSM is responsible for sequencing and controlling the processes in the AES design's datapath. For the security assurance of this cryptographic module, the state transactions should happen in the appropriate order as defined in the associated standard AES algorithm. The STG of the control FSM extracted by the *RTL-FSMx* tool [19] from its RTL description is shown in Fig. 5. According to the standard AES encryption algorithm, the state 'DO_ROUND' must be reached before the 'FINAL_ROUND' state, which is evident from the extracted STG of the FSM.

Nevertheless, it is critical to validate whether this order of states of the control FSM (DO_ROUND $\rightarrow$ FINAL_ROUND) is maintained in the gate-level netlist implementation obtained after logic synthesis. The primary motivation is that in the gate-level netlist abstraction, hidden don't-care transitions can be introduced by the state-of-the-art commercial logic synthesizers [18]. Such don't-care state transitions are absent in the RTL description of a control FSM. Due to such transitions, an attacker can bypass the specified order of the states, which was supposed to be followed rigorously. If, due to such an improper implementation of the FSM, the mentioned order of states is not held, then it may aid an attacker in reaching the 'FINAL_ROUND' state directly from the other states, such as from the reset state. Herefore, the encryption operation will not be effective or can eventually lead to the encryption operation's intermediate result or key leakage. Additionally, the output result of the encryption might be corrupted. As a result, the

performance, security, and reliability of the entire system might be affected [1], [18]. Therefore, in such a scenario, this security rule will be violated, and a warning message will be provided to the designer to indicate the existence of this security flaw. The designer can clear this bug by taking appropriate countermeasures, such as tweaking the RTL design to ensure that the specified order of critical states is always maintained at the gate-level netlist abstraction.

R5: *Dead (inactive) states should be absent in the extracted state transition graph (STG) of a control FSM from the netlist*. In this paper, the dead state of an FSM is defined as a state which never gets visited and has no input and output transition conditions. Sometimes, this state may have a self-transition scenario. In the STG shown in Fig. 6, the state '010' is dead. Since a dead state can never be triggered using any input transition condition, such a state remains undetected if a simulation-based approach is used. The presence of dead states in the extracted STG of a control FSM is potentially dangerous from a security standpoint. It is because such an undetectable state may aid an attacker in inserting a stealthy Trojan utilizing that state. This scenario may lead to information leakage or access control issues [1]. Removing the dead states can readily fix such a security-critical bug.

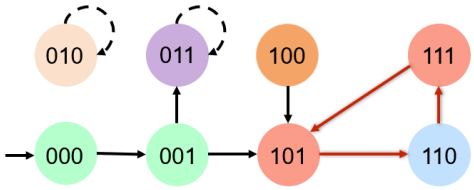

Fig. 6: The STG of a particular control FSM with potential structural issues from the hardware security standpoint.

R6: *Unreachable states with a transition to a non-reset state should not exist in the obtained STG of a control FSM from the netlist*. In this paper, we define the unreachable state of an FSM as a state without any input transition condition but having single or multiple transitions to other states. In Fig. 6, the state '100' is unreachable. The presence of unreachable states in the STG with a transition to a non-reset state indicates the existence of a security-critical structural flaw. It is dangerous from the security perspective since an attacker can insert a malicious Trojan utilizing this state to access the protected states of an FSM, which are crucial to the designers. Therefore, the existence of such unreachable states indicates the presence of a security bug in the FSM design. This issue should be detected and resolved as soon as possible. The designers will receive warnings on such a bug if this security rule is violated. The designers can either remove those unreachable states if unnecessary, take necessary steps to cause the unreachable states to transit to the reset state only, or make those reachable by other states of the FSM to fix such a bug in the control FSM implementation. However, sometimes, a particular unreachable state may only transit to the reset state of an FSM, and it is not dangerous from a security perspective. For instance, if the 'default' statement is used to cover the unused states of an FSM in the RTL description, then such a scenario will be found if the extracted gate-level STG of the FSM is analyzed. This rule violation will not occur in such a case, and a warning will not be provided to the designer.

R7: *States with static deadlock conditions should not be present in the extracted STG of a control FSM from the netlist*. The state of a control FSM is said to be in static deadlock condition if there exist only entry points to the state but no exit point from that state. The state may have a self-transition condition. As shown in Fig. 6, the '011' state has this characteristic. The control FSM freezes upon entering such a state. Hence, it is potentially dangerous from the hardware security point of view because such a state may aid an attacker in exploiting this state and thus launching a successful DoS attack. This security-critical flaw should be detected and fixed as early as possible. The designers can resolve this issue by transitioning to a secure state from the state with a static deadlock scenario upon encountering a warning on the violation of security rule R7.

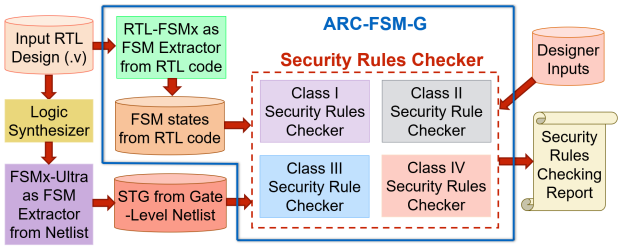

Fig. 7: Overview of the ARC-FSM-G framework.

R8: *Group of states with a dynamic deadlock loop that includes a protected state should be absent in the extracted STG of a control FSM*. Apart from the states with static deadlock scenarios, certain states of the control FSM may form a group with a dynamic deadlock loop. Such a group of states may have single or multiple entering paths; however, it does not have any exit from that group. Once entering the group, the control FSM circulates in the loop containing only those states, and there is no exit point from the loop. This situation is dangerous from a security standpoint if the group includes one or more protected states specified by the designer, and such a group is undesirable to the designer. As illustrated in Fig. 6, the group of states {*'101', '110', '111'*} is such an example, where '110' is a protected state. The presence of such a group may indirectly assist an attacker in getting access to the protected state by somehow entering the group once the FSM starts running in the dynamic deadlock loop. Reverting to the FSM's initial (reset) state is impossible without applying a hardware reset. Hence, the designers should be able to identify and fix this sort of security bug as early as possible. The designers can create a transition to a secure state from the group of states existing in such a dynamic deadlock loop for a quick fix to satisfy the security rule R8.

V. ARC-FSM-G Framework

The high-level overview of our proposed ARC-FSM-G framework is illustrated in Fig. 7. As shown in the figure, there are three primary inputs to the ARC-FSM-G framework. The first input is the RTL design written in HDL (e.g., VHDL or Verilog) to be analyzed. The ARC-FSM-G framework incorporates *RTL-FSMx* [19] as an integral and constituting module, a fast and accurate control FSM extractor from RTL code. *RTL-FSMx* is a static analysis-based solution, and the framework analyzes the input high-level RTL code to extract the states of the control FSMs with the associated state encoding values. The second input to the ARC-FSM-G framework is the gate-level STGs of the control FSMs existing in the input RTL design. The input RTL design is first synthesized. Next, the

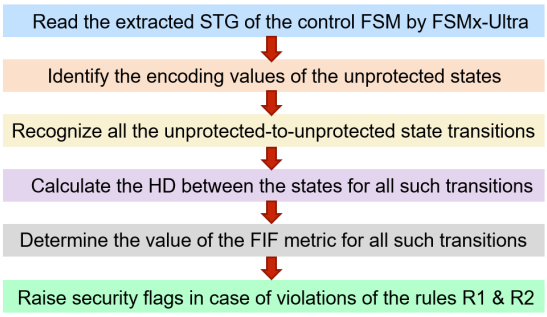

Fig. 8: Algorithmic flow of the Class I Security Rules Checker.

obtained gate-level netlist is analyzed using the techniques like *FSMx-Ultra* framework [20] to recognize and isolate the control FSMs precisely from other registers and extract the corresponding gate-level STGs of the FSMs.

The *FSMx-Ultra* framework also yields *FSM Register Candidates Report*, which contains all the names and sizes of the FSM state registers extracted from the gate-level netlist with other relevant information. This secondary output aids our proposed ARC-FSM-G framework to map the obtained gate-level STGs with the corresponding FSM state registers from the RTL description extracted by *RTL-FSMx* [19]. It is because the naming of the registers is conserved in the logic synthesis stage of the input RTL design [18]. For instance, if an FSM state register is defined as 'reg [2:0] state' in the RTL code, then the RTL description is transformed into 3 flip-flops with the names 'state_reg[0], state_reg[1], and state_reg[2]' after logic synthesis by the state-of-the-art commercial logic synthesis tools. These 3 flip-flops form a register named 'state_reg' in the obtained gate-level netlist. Our framework utilizes this fact for the successful mapping process. Moreover, ARC-FSM-G checks whether the STG extracted from the RTL description using RTL-FSMx [19] is a subset of the gate-level STG yielded by FSMx-Ultra [20] to verify the control-flow equivalence of the FSM state register under assessment. Such a structural validation phase strengthens the mapping stage further. This mapping phase is the crucial stage which makes the connecting bridge between the information of the control FSMs obtained from the RTL description through static analysis performed by *RTL-FSMx* and their associated gate-level STGs extracted by *FSMx-Ultra*. Finally, after mapping, the *Security Rules Checker* module analyzes the gate-level STGs to validate whether the proposed security rules mentioned in Table I for FSM design are followed.

The third and final one is several user inputs defined by the designer. These inputs include the list of protected states, authorized states, and the specific order of visiting two states that must be maintained. The designer can provide this information for single or multiple control FSMs. The ARC-FSM-G framework generates a comprehensive security rules-checking report after analysis. This report warns the designers of existing security bugs in the gate-level netlists of the FSMs of the input RTL design in case any of the proposed security rules are violated. Then the designers can review the obtained report to get an idea about the security-critical bugs and fix those according to the sample guidelines presented in Section IV. The rule violation detection schemes for the classes mentioned in Table I are presented below and implemented by the *Security Rules Checker* module. It needs to be noted that the ARC-FSM-G framework gets aware of the control FSMs present in the design and their associated gate-level STGs after the mapping process mentioned before. Hence, this mapping stage is crucial in the further analysis phases of our proposed framework. These analysis phases are performed on the extracted gate-level STGs of the control FSMs one after another and considering one at a time.

Class I: The *Class I Security Rules Checker* module checks for the violations of 2 security rules (R1 and R2) as discussed in Section IV. After the mapping phase, for each of the detected control FSMs by the *FSMx-Ultra* framework [20], the algorithmic flow illustrated in Fig. 8 is implemented. The rules violation detection phase starts with reading the yielded STG of the control FSM by *FSMx-Ultra*. Next, all the unprotected states of the control FSM description are identified. The set of protected states specified by the designer is considered in this process. Then, all the state transitions between two unprotected states are recognized. Finally, the *Hamming Distance (HD)* is calculated for all such transitions. If HD is greater than 1, the transition is marked as potentially dangerous, and a security flag will be raised later to warn the designers. Moreover, the values of the *FIF* metric for all the unprotected-to-unprotected state transitions of a control FSM are found using Eq. 1 with the extension mentioned in Section IV. Suppose the *FIF* metric value is 1 for such a particular state transition. In that case, the transition is potentially vulnerable to setup time violation-based fault-injection attacks, and a security flag will be raised. In this manner, the two security flags are raised individually.

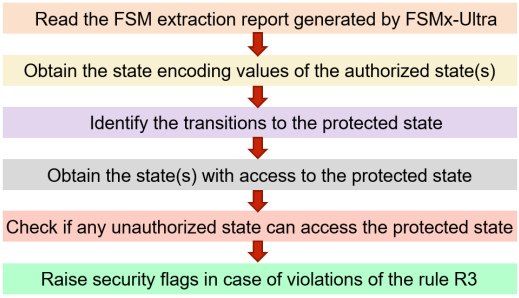

Fig. 9: Algorithmic flow of the Class II Security Rule Checker.

Class II: The *Class II Security Rule Checker* module aims to check whether any of its protected states is accessible by an unauthorized state of an FSM netlist. Its algorithmic flow to detect the violations of rule R3 is presented in Fig. 9. First, the rule checker module reads the FSM extraction report generated by FSMx-Ultra. Secondly, the state encoding values of the authorized and protected states specified by the designer as the input are obtained. After successful mapping, the ARC-FSM-G framework knows the associated protected and authorized states of a particular control FSM. Thirdly, the transitions to the protected states are identified, which were specified by the designer. Finally, it is checked if any unauthorized state can access a particular protected state by analyzing the incoming transitions of that state, thus raising the security flag to point out that rule R3 has been violated. Such a flag will warn the designers of security bugs in the FSM netlist.

Class III: The *Class III Security Rule Checker* module focuses on validating whether the sequence of the states specified by the designer is rigorously maintained in the gate-level implementation of the FSM. The algorithmic flow of the module is illustrated in Fig. 10. The analysis phase starts with reading the extracted gate-level STG of a particular

control FSM. The designer can specify the order of the states using the symbolic representation of the FSM states present in the high-level RTL description. Hence, after the mapping phase, the ARC-FSM-G framework obtains the associated encoding values of the states existing in the mentioned order in binary format. A particular specified order is deconstructed into source and destination states. The source state must be visited first before transiting to the destination state. The designer can also provide state-visit orders for multiple control FSMs or multiple orders for a single FSM as input to the proposed ARC-FSM-G framework. Then, a directed graph representation of the STG is generated. Next, all the paths from the reset state to the destination state are obtained using the *Depth-First Search (DFS)* algorithm. Finally, the obtained paths are analyzed to detect if the source state of the state-visit order under analysis is always present in the path. Since the STG is directed in nature, this fact ensures that the source state is always visited before reaching the destination state. If the source state is not present in such a path, it is marked as a security-critical bug, and a security flag is raised. It will warn the designer about the violation of R4, and the path violating the rule with the missing source state will be reported.

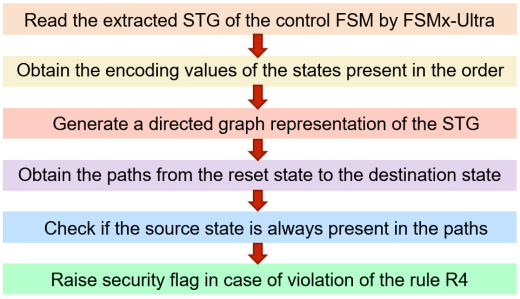

Fig. 10: Algorithmic flow of Class III Security Rule Checker.

Class IV: The primary purpose of the *Class IV Security Rules Checker* module is to check for structural issues in the extracted gate-level STGs of the control FSMs present in the netlist and warn the designers accordingly. The algorithmic flow of this module is presented in Fig. 11. The security rules-checking process initiates with reading the extracted gate-level STGs of the associated control FSMs by FSMx-Ultra one at a time. Next, the module creates a directed graph representation of the extracted gate-level STG to perform appropriate structural analysis on the STG. Then, the dead states present in the STG are identified. The ARC-FSM-G framework marks a particular state of an FSM as a dead state if all of these 3 requirements are met: (i) no transition to it from the other states, (ii) may or may not have a self transition, and (iii) no transition to the other states. Moreover, the dangerous unreachable states of the FSM are detected. These 3 conditions must be satisfied to label such a particular state of the FSM under assessment: (i) no transition from the other states, (ii) may or may not have a self transition, and (iii) has single or multiple transitions to the other non-reset states.

Finally, the module recognizes the states with static deadlock conditions and the groups of states with dynamic deadlock loop that includes one or multiple protected states. A particular state of an FSM is treated as a state with static deadlock condition if it satisfies these 3 prerequisites: (i) has at least one incoming transition from the other states, (ii) may or may not have a self transition, and (iii) no transition to

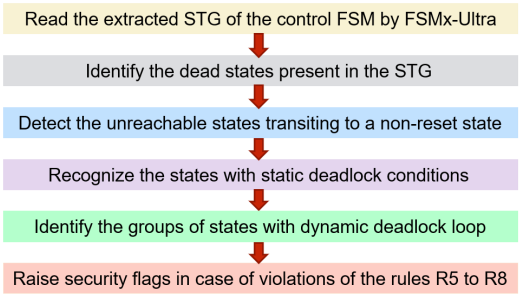

Fig. 11: Algorithmic flow of Class IV Security Rules Checker.

the other states. The mentioned requisites for detecting the dead, unreachable, and states with static deadlock scenarios can be checked by analyzing a particular state's incoming and outgoing transitions. Last but not least, the group of states forming a loop and containing single or multiple protected states are identified using [21], and it is checked whether such a group has these 2 characteristics: (i) single or multiple incoming transitions to at least a state of the group from the other states, and (ii) no outgoing transition. Security flags are raised to warn the designers about violations from R5 to R8, depending on the existence of such states or groups of states.

In summary, the *RTL-FSMx* [19] module of our proposed ARC-FSM-G framework performs static analysis on the input RTL source codes of the design to recognize the control FSMs with their associated states. Then, the *Security Rules Checker* module performs appropriate mathematical and graph algorithmic analysis on the gate-level STGs of the control FSMs extracted by *FSMx-Ultra* [20] to detect the violations of the proposed security rules (R1 to R8) in this paper. The ARC-FSM-G framework generates a comprehensive security rules-checking report based on the security flags raised in case of violations. This report warns the designers of potentially dangerous bugs from the hardware security perspective. Such security bugs can be fixed according to the sample guidelines discussed in Section IV to satisfy the proposed rules for secure control FSM implementation at the netlist abstraction.

VI. Experimental Results

A. Complexity of ARC-FSM-G

The algorithmic complexity of the proposed ARC-FSM-G framework can be evaluated by analyzing its overall time complexity and space (memory) complexity. As presented in Section V, the ARC-FSM-G framework incorporates the RTL-FSMx framework with the *Security Rules Checker* module. Hence, both of these components contribute to the overall algorithmic complexity of our proposed ARC-FSM-G framework. RTL-FSMx requires storing the abstract syntax tree (AST) representation of the entire RTL source code in the memory, a tree-like data structure having numerous nodes. The time complexity of RTL-FSMx is $O(M)$ approximately, where M is the total number of 'always' blocks present in the input RTL source code. Moreover, the space complexity of RTL-FSMx is $O(N)$ where N is the total number of nodes of the AST [19]. Finally, the overall algorithmic complexity of the *Security Rules Checker* module is primarily dominated by the complexity of the *Johnson's Algorithm* [21] since this analysis phase is the most intensive one in terms of computation and memory. Therefore, the overall time complexity of this module is $O((V + E)(c + 1))$ roughly, where V and E are the total

TABLE II: FSM extraction results for 15 benchmarks obtained from [23]–[25] and other resources using [19] and [20].

Benchmark Name	Line Count	'always' Block Count	Gate Count	FF Count	Control FSM Count	State Register Size	RTL STG State Count	GL STG State Count	GL STG Edge Count
I2C CORE	1263	22	1451	125	2	18, 6	18, 6	752, 64	764, 109
SAYEH CPU	1280	12	3218	170	1	4	11	16	72
SUBTERRANEAN	1222	5	4800	263	1	6	33	64	66
XTEA CIPHER	601	6	5095	105	1	2	4	4	6
ROMULUS	816	8	8977	585	1	4	9	16	28
GIFT-COFB	917	5	9237	337	1	3	7	8	21
ASCON AED	1571	7	9803	459	1	6	44	64	87
MEMORY CONTROLLER (V1)	5824	175	9843	992	1	7	66	128	334
MEMORY CONTROLLER (V2)	5824	175	10057	1051	1	66	66	133	155
PICORV32 CPU	3044	39	17101	1680	3	3, 2, 2	8, 4, 3	8, 4, 4	32, 8, 11
SAEAES	1694	9	23329	403	2	2, 4	3, 9	4, 16	7, 42
COMET	1747	8	24439	530	2	2, 4	3, 12	4, 16	7, 46
SHA-512	1485	16	45142	3673	1	2	3	4	7
GCM AES (V1)	1760	39	45192	1692	1	4	10	16	28
GCM AES (V2)	1760	39	45402	1696	1	10	10	144	147

numbers of nodes and edges of the gate-level STG of an FSM, respectively, and c is the number of cycles (loops) in the gate-level STG. Furthermore, the overall time complexity of the rules checker module can be approximated as $O(V^2)$, where V is the total node count of the gate-level STG of an FSM. The overall algorithmic complexity of the ARC-FSM-G framework combines the associated complexities of its two component modules. It can be roughly determined by the complexity of the graph algorithmic analysis performed on the gate-level STGs of the control FSMs using the *Johnson's Algorithm* [21] since the complexity of the RTL-FSMx module is negligible.

B. Performance of ARC-FSM-G

We have analyzed 15 RTL benchmarks from various open-source repositories [23]–[25] and other resources to evaluate the efficacy of our proposed ARC-FSM-G framework. These benchmarks vary in size, complexity, and functionality (cryptographic cores, processor cores, communication controllers, etc.). The total number of lines and the total count of 'always' blocks (M) existing in the high-level RTL codes are shown in Table II. These RTL designs were synthesized using *Design Compiler* [26], and *SAED90nm* was used as the target technology library. However, *Genus* from *Cadence* can be also used as the logic synthesizer. The associated gate count and flip-flop count can be obtained from the generated synthesis report. The total count of control FSMs with their associated state register size was obtained from the FSM extraction report generated by RTL-FSMx [19]. Moreover, the total number of states present in the STG extracted from the RTL description (RTL STG) was also obtained from that report. These states are the user-defined states existing in the RTL description of the control FSMs. As evident from Table II, the chosen benchmarks also vary widely in terms of control FSM count and the state-space complexity of the control FSMs, which is determined by the size of the corresponding FSM state registers. We have also used *SpyGlass* [6] from *Synopsys* to ensure that the RTL benchmarks are free from pre-synthesis lint issues.

After performing logic synthesis and optimization, the FSMx-Ultra framework proposed in [20] was used to extract control FSMs with the associated gate-level STGs from the synthesized gate-level netlists. The number of states (V) and the number of edges (E) were obtained by analyzing the gate-level STGs (GL STGs) generated by FSMx-Ultra. The state encoding schemes employed with the corresponding state register size of the control FSMs of the chosen 15 benchmarks are illustrated in Table III. FSMx-Ultra can successfully recover the control-flow of the FSM under assessment from the synthesized gate-level netlist with hidden don't-care states and transitions, even for very large netlists [20]. However, it needs to be noted that as the size of the FSM register (n) increases, its state-space complexity grows exponentially (2^n). Therefore, when a particular control FSM is encoded using the *One-Hot* encoding scheme, it results in a massive number of don't-care states. Extracting all these possible don't-care states is practically infeasible for very large FSMs. In such a scenario, for a large FSM utilizing the *One-Hot* encoding approach, FSMx-Ultra yields a partial gate-level STG containing the control-flow specified in the RTL description with some other don't-care states and transitions instead of a complete STG.

Nonetheless, from the hardware security perspective, not all don't-care states of a security-critical control FSM are dangerous [1], [18], [20]. The don't-care states having direct access to the designer-defined states present in control FSMs' RTL description are potentially dangerous. The underlying reason is that if a don't-care state can access a protected state directly, an attacker can inject fault to transit to that particular don't-care state by altering the voltage supply, clock frequency, or temperature, and these types of attacks require low-cost equipment. Therefore, the presence of such don't-care states poses the most severe security threat [1], [18]. To conclude, the massive state-space of a large FSM using the *One-Hot* encoding technique can be shrunken, and analysis of the partial gate-level STG obtained by FSMx-Ultra will be sufficient. Hence, the ARC-FSM-G framework does not require extracting the complete gate-level STG with all don't-care states of a large *One-Hot* encoded control FSM.

We have developed an automated tool implementing the ARC-FSM-G framework using *Python*. The overall run-time of the tool is presented in Table III, which incorporates the run-times of its two constituting modules. All the experiments on the benchmark designs have been performed using an Intel Core i7-1065G7 CPU clocked at 1.3 GHz with 16GB RAM on a personal desktop. Finally, we have used the property-based formal verification approach using *JasperGold* to validate the results obtained by the ARC-FSM-G framework. The security properties were written according to [22], and the same rule violations were detected using the formal approach. The results prove that ARC-FSM-G can effectively detect the security bugs indicating the associated security rule violations at the netlist abstraction within a few seconds. Last but not least, it is apparent from Table III that our proposed framework requires a low peak memory usage during the run-time of the tool. Hence, ARC-FSM-G is both run-time and memory efficient.

C. Case Studies

We present case studies on 2 practical benchmarks chosen from [23], [25], which were analyzed by our proposed ARC-FSM-G framework. These case studies illustrate that ARC-FSM-G can effectively detect violations of the proposed 8 security rules within a few seconds while requiring a low peak memory usage. One benchmark is the complex *Memory*

TABLE III: Run-time and peak memory usage of ARC-FSM-G for 15 benchmarks obtained from [23]–[25] and other resources.

Benchmark Name	State Register Size	State Encoding Scheme	Protected State	Authorized State	State Visit Order	Violated Rules	Run-time (s.)	Peak Memory (MB)
I2C CORE	18	One-Hot	wr_a	idle	idle → wr_a	R1, R6, R7	4.894	32.870
	6	One-Hot	ST_WRITE	ST_START	ST_START → ST_WRITE	R1, R3, R4, R6		
XTEA CIPHER	2	Binary	CTRL_ROUNDS1	CTRL_ROUNDS0	CTRL_ROUNDS0 → CTRL_ROUNDS1	R1, R2	2.147	12.183
SAYEH CPU	4	Binary	memread	fetch	fetch → memread	R1, R2, R3	2.553	12.184
SUBTERRANEAN	6	Binary	Mabs	ADabs	ADabs → Mabs	R1, R2	2.846	12.183
MEMORY CONTROLLER (V1)	7	Binary	BG0	IDLE	IDLE → BG0	R1, R2, R5, R6, R7	4.185	19.771
MEMORY CONTROLLER (V2)	66	One-Hot	BG0	IDLE	IDLE → BG0	R1, R6, R7	5.615	18.329
ROMULUS	4	Binary	Mstagei	Nonceonly	Nonceonly → Mstagei	R1, R2, R3	2.221	12.184
ASCON AED	6	Binary	Finalization1	ciphertext6	ciphertext6 → Finalization1	R1, R2, R3, R4	3.085	12.183
GIFT-COFB	3	Binary	waitCBMessage	Message	Message → waitCBMessage	R1, R2, R3, R8	2.242	12.184
PICORV32 CPU	3	Binary	cpu_state_stmem cpu_state_ldmem	cpu_state_ld_rs1 cpu_state_ld_rs2	cpu_state_ld_rs1 → cpu_state_stmem cpu_state_ld_rs2 → cpu_state_stmem cpu_state_ld_rs1 → cpu_state_ldmem	R1, R2, R3, R4	4.388	12.202
	2	Binary	1	0	0 → 1	R1, R2		
	2	Binary	WBEND	WBSTART	WBSTART → WBEND	No Violation		
SHA-512	2	Binary	CTRL_DONE	CTRL_ROUNDS	CTRL_ROUNDS → CTRL_DONE	R5	2.245	12.184
COMET	2	Binary	CTRL_EXEC	CTRL_INIT	CTRL_INIT → CTRL_EXEC	No Violation	2.776	12.184
	4	Binary	ProduceT2	ProduceT	ProduceT → ProduceT2	R1, R2		
SAEAES	2	Binary	CTRL_EXEC	CTRL_INIT	CTRL_INIT → CTRL_EXEC	No Violation	2.751	12.184
	4	Binary	Tprocess	waitT, waitNonce	waitNonce → Tprocess	R1, R2		
GCM AES (V1)	4	Binary	M_ENCRYPT	INC_COUNTER	INC_COUNTER → M_ENCRYPT	R1, R2	3.189	12.184
GCM AES (V2)	10	One-Hot	M_ENCRYPT	INC_COUNTER	INC_COUNTER → M_ENCRYPT	R1, R7	4.416	14.343

Controller IP core from [25], and the other one is a 32-bit processor based on the RISC-V architecture, *PICORV32* [28].

1) Memory Controller: The *Memory Controller* IP core from [25] can be used in numerous embedded system applications. The core supports SDRAM, SSRAM, FLASH memory, ROM, and several other devices. It has 8 chip select signals in total, and each of those is programmable. Moreover, the controller provides default boot sequence support with other necessary features [27]. The IP has a single control FSM, as evident from Table II. We have created a version (V1) of this IP core utilizing the *Binary* encoding scheme. The IP core contains 66 states, so it requires a control FSM having a state register with 7 flip-flops. The size of the state register will remain unchanged if the *Gray* encoding scheme is employed. The total count of states and edges in the gate-level STG extracted by FSMx-Ultra are 128 and 334, respectively. ARC-FSM-G analyzed this massive gate-level STG, and 5 security rule violations were detected (R1, R2, R5, R6, R7), as presented in Table III. The overall run-time of the tool implementing the framework was only 4.185 s., and the peak memory usage during run-time was only 19.771 MB.

It needs to be noted that, as shown in Table III, the state of the FSM, named 'BG0', was considered a protected state. The reason is that an external bus master can access the memory bus if the FSM is in this state. It is a crucial state from the security perspective since the memory bus may contain sensitive data. However, the external bus master must grant access by asserting the 'mc_gnt' signal, which is only checked in the 'IDLE' state. Therefore, the 'IDLE' state is authorized, and the state order 'IDLE → BG0' must be maintained. It is because an attacker may inject a fault in the system and transit to the 'BG0' state directly, which permits access to the memory bus bypassing the assertion checking of the 'mc_gnt' signal. It can eventually lead to information leakage issues. The ARC-FSM-G tool generates an annotated gate-level STG of the control FSM illustrating the identified security rule violations and the textual report. Therefore, it can help the designers to recognize security bugs quite rapidly.

Finally, the V2 version of the *Memory Controller* core implementing the original design from [25] was analyzed by ARC-FSM-G. In this version, the control FSM is encoded using the *One-Hot* scheme, and hence the 66 states of the FSM require a state register having 66 flip-flops. This huge FSM comes with a massive state space, i.e., 2^{66} states, where only 66 states are described in the RTL description. Therefore, the number of don't-care states is enormous. Nevertheless, only the don't-care states connected with the control-flow specified in the RTL abstraction are crucial from the security perspective as those pose the most severe threat. Hence, the gate-level STG extracted by the FSMx-Ultra framework [20] yields 133 states and 155 state transitions in total instead of 2^{66} states (which is practically infeasible to obtain), as evident from Table II. The number of don't-care states present in the extracted gate-level STG is 67. The ARC-FSM-G framework analyzed this enormous control FSM STG, and violations of 3 security rules (R1, R6, and R7) were reported (see Table III). The tool required only 5.615 s. and peak memory usage of 18.329 MB to generate the report with the huge annotated STG.

2) PICORV32: The *PICORV32* is a RISC-V-based processor (CPU) core that implements the *RISC-V RV32IMC Instruction Set* [28]. This CPU core can be configured in 5 different modes of operation: *RV32E, RV32I, RV32IC, RV32IM, or RV32IMC*. It optionally contains a built-in interrupt controller. The core has a high f_{max} of around 250-450 MHz on a 7-Series Xilinx FPGA. The CPU core can be an auxiliary processor in ASIC and FPGA designs. Moreover, it can be integrated into most existing designs without crossing clock domains because of having a high f_{max} [28]. As presented in Table II and Table III, the processor has 3 control FSMs, and all the FSMs were encoded using the *Binary* encoding scheme. The sizes of the FSM state registers are 3, 2, and 2, respectively. As shown in Table III, multiple protected states, authorized states, and state visit orders can be specified.

The first 2 control FSMs of this CPU include memory read and write states, which should be only accessed by the memory instructions. We assume that the privilege control is implemented in the software. The control mechanism analyzes the memory instructions from the user kernel. A flag is raised if the mechanism does not access memory locations dedicated to the system kernel. An attacker's objective might be to inject a fault in the system during non-memory instruction execution and bump to the states that control memory read and/or write

operations. As a result, the implemented privilege control can be bypassed, and the memory locations of the system can be accessed illegally. Therefore, the memory read and/or write controlling states of these 2 control FSMs are protected states. For the last control FSM, the 'WBEND' state indicates that the data transfer operation to the system data bus is completed. An attacker can access this state illegally by performing a fault-injection attack to provide security-sensitive data from the memory to the system data bus. Hence, 'WBEND' is treated as the protected state. Similar intuition was used for identifying the security-critical states and thus selecting the protected states of other benchmarks presented in Table III.

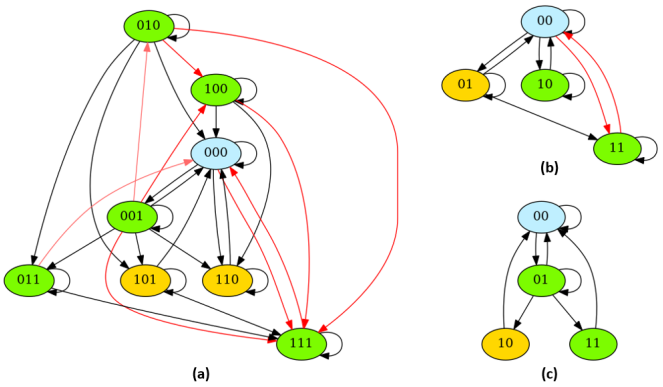

Fig. 12: Annotated gate-level STGs of *PICORV32*'s FSMs.

The ARC-FSM-G tool analyzed the RTL design of the processor core and the gate-level STGs yielded by FSMx-Ultra [20]. Security rules violations were detected in the first 2 control FSMs. 4 security rules (R1, R2, R3, and R4) were violated in the first FSM design, whereas only 2 rule violations (R1 and R2) were found in the second one. However, no security rule violations occurred in the third control FSM. As mentioned in Table III, the overall run-time of the ARC-FSM-G tool was only 4.388 s. to detect all such violations, and the peak memory usage during run-time was only 12.202 MB. The annotated gate-level STGs of the 3 control FSMs generated by the tool are illustrated in Fig. 12. The annotated gate-level STGs of the 3 control FSMs, as shown in Fig. 12, correspond to the FSMs driven by the state variables 'cpu_state', 'mem_state' and 'state', respectively, which are also reported by the ARC-FSM-G tool. It helps to pinpoint the FSM with potential security issues at the netlist abstraction specified in the RTL code. The tool also provides the flexibility to omit an FSM from the security rules-checking process. The designer must provide at least a protected state, an authorized state, and a state visit order of an FSM for assessment by ARC-FSM-G. Otherwise, the FSM will not be assessed since all FSMs might not be security-critical. In Fig. 12, the states colored in sky blue are the reset states of the FSMs. The ones with lime green and golden colors are normal and protected, respectively. Different color coding is used for visualizing the states violating the rules R5 to R8. The transitions marked in red violate the rules R1 or R2 to indicate potentially vulnerable state transitions. We used *PyGraphviz* library for visualizing the annotated gate-level STGs of the control FSMs.

VII. Conclusion and Future Work

This paper proposes a set of security rules for secure control FSM design to be validated after the logic synthesis stage. These rules will assist in making the control FSM design resilient against certain fault-injection and denial-of-service attacks. Moreover, the proposed security rules will help to ensure that FSM is free from potential information leakage and access control issues. Additionally, this paper presents an automated verification framework named ARC-FSM-G to detect violations of the proposed rules. We have also provided guidelines on how to satisfy the rules in case of violations. The ARC-FSM-G framework makes the designers aware of potentially dangerous security bugs present in the gate-level implementation of a control FSM. Then, the designers can take appropriate countermeasures upon reviewing the security rule-checking report. Results on several benchmarks varying in size and complexity have proved the efficacy of ARC-FSM-G. However, an exhaustive analysis of the effects of satisfying the security rules presented in this paper is one of our future research directions. Finally, we envision extending the proposed security rules set and developing such rules for the datapath logic of an SoC at the netlist abstraction.

References

[1] A. Nahiyan et al.,"AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," *Design Automation Conference*, pp. 1-6, 2016.
[2] M. Tehranipoor and F. Kaushanfar, "A survey of hardware Trojan taxonomy and detection," *IEEE Design and Test of Computers*, 2010.
[3] G. K. Contreras et al.,"Security vulnerability analysis of design-for-test exploits for asset protection in SoCs," *ASP-DAC*, pp. 617-622, IEEE, 2017.
[4] P. Mishra et al,"Hardware IP security and trust," *Springer*, 2017.
[5] D. M. Anderson,"Design for manufacturability: how to use concurrent engineering to rapidly develop low-cost, high-quality products for lean production," *CRC press*, 2020.
[6] https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html
[7] J. He et al.,"SoC interconnection protection through formal verification," *Integration 64*, pp. 143-151, 2019.
[8] J. Sepulveda et al.,"Towards the formal verification of security properties of a Network-on-Chip router," *2018 IEEE 23rd ETS*, pp. 1-6, 2018.
[9] N. Farzana et al.,"SoC Security Verification using Property Checking," *2019 IEEE International Test Conference (ITC)*, pp. 1-10, 2019.
[10] A. Ardeshiricham et al.,"Register transfer level information flow tracking for provably secure hardware design," *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1691–1696, 2017.
[11] C. Brant et al.,"Challenges and Opportunities for Practical and Effective Dynamic Information Flow Tracking," *Comput. Surveys 55, 1*, 2021.
[12] S. K. Muduli et al.,"Hyperfuzzing for SoC security validation," *39th International Conference on Computer-Aided Design (ICCAD)*, 2020.
[13] A. Tyagi et al.,"TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," *arXiv:2201.09941 [cs]*, 2022.
[14] M. Mushtaq et al.,"WHISPER: A Tool for Run-Time Detection of Side-Channel Attacks," *IEEE Access 8*, 2020.
[15] S. Johnson,"Lint, a C Program Checker," *Computer Science Technical Report*, 1978.
[16] https://cwe.mitre.org/data/definitions/1245.html, "CWE-1245".
[17] L. Zussa et al.,"Investigation of timing constraints violation as a fault injection means," *Design of Circuits and Integrated Systems*, 2012.
[18] A. Nahiyan et al., "Security-Aware FSM Design Flow for Identifying and Mitigating Vulnerabilities to Fault Attacks," *IEEE TCAD*, 2019.
[19] R. Kibria et al.,"RTL-FSMx: Fast and Accurate Finite State Machine Extraction at the RTL for Security Applications," *2022 IEEE International Test Conference (ITC)*.
[20] R. Kibria et al.,"FSMx-Ultra: Finite State Machine Extraction from Gate-Level Netlist for Security Assessment," *IEEE TCAD*, 2023.
[21] Donald B. Johnson,"Finding All the Elementary Circuits of a Directed Graph," *SIAM Journal on Computing*, 1975.
[22] https://trust-hub.org/#/data/Security-Properties-Rules, "Trust-Hub".
[23] https://github.com/freecores, "FreeCores".
[24] https://github.com/secworks, "SecWorks".
[25] https://opencores.org/, "OpenCores".
[26] "Design Compiler® Optimization Reference Manual – Version F-2011.09-SP2," Synopsys®, December 2011.
[27] https://github.com/freecores/mem_ctrl, "Memory Controller IP".
[28] https://github.com/YosysHQ/picorv32/blob/master/picorv32.v.