FSMx-Ultra: Finite State Machine Extraction from Gate-Level Netlist for Security Assessment

Rasheed Kibria, Farimah Farahmandi, *Member, IEEE*, and Mark Tehranipoor, *Fellow, IEEE*

***Abstract*—Numerous security vulnerability assessment techniques urge precise and fast finite state machines (FSMs) extraction from the design under evaluation. Sequential logic locking, watermark insertion, fault-injection assessment of a System-on-a-Chip (SoC) control flow, information leakage assessment, and reverse engineering at gate-level abstraction, to name a few, require precise FSM extraction from the synthesized netlist of the design. Unfortunately, no reliable solutions are currently available for fast and precise extraction of FSMs from the highly unstructured gate-level netlist for effective security evaluation. The major challenge in developing such a solution is precise recognition of FSM state flip-flops in a netlist having a massive collection of flip-flops. In this paper, we propose *FSMx-Ultra*, a framework for extracting FSMs from extremely unstructured gate-level netlists. *FSMx-Ultra* utilizes state-of-the-art graph theory concepts and algorithms to distinguish FSM state registers from other registers and then constructs gate-level state transition graphs (STGs) for each identified FSM state register using automatic test pattern generation (ATPG) techniques. The results of our experiments on 14 open-source benchmark designs illustrate that *FSMx-Ultra* can recover all FSMs quickly and precisely from synthesized gate-level netlists of diverse complexity and size utilizing various state encoding schemes.**

***Index Terms*—FSM Automata Theory, FSM Extraction, Netlist Analysis, Security Assessment**

I. Introduction

Modern *System-on-a-Chip (SoC)* designs are sophisticated entities and primarily composed of several functional units known as hardware *Intellectual Property (IP)* cores that interact with each other and collaborate to accomplish complex tasks and provide the desired functionality. To reduce the overall expenses and shorten the *time-to-market (TMT)* as much as possible, the design firms extensively rely on third-party vendors to develop, implement, integrate, and fabricate their IP designs. As a result, the designer's IPs get transparent to numerous untrusted stakeholders. Therefore, IPs eventually become vulnerable to tampering attacks [1] and IP infringement [28]. Furthermore, researchers have shown that SoC security may be at risk when deployed in operation [2]. Attackers may use the *design for test (DFT)* structures to their advantage or perform power, timing, and electromagnetic emission-based analysis, inject faults to access the system illegally, or leak sensitive and secret information such as the keys used in cryptographic encryption and decryption [3]–[6].

Numerous security assessment techniques have been proposed to evaluate and address the aforementioned threats over the past years. These techniques have primarily concentrated on protecting the device's control flow, which is crucial to the entire system's operation. Since control logic units are typically FSM-based, such techniques frequently require precise recognition of all *finite state machine (FSM)* structures. For instance, when translating from RTL to gate-level abstraction, the *Computer-Aided Design (CAD)* tools may add more don't-care states to the design's control FSM. Attackers might use fault-injection techniques to gain access to the design's protected states via utilizing the don't-care states [45]. Furthermore, while integrating the *DFT* structures at the gate-level abstraction, untrustworthy *third-party IP (3PIP)* vendors can implant sequential *Trojans* into the design's control FSM [7]–[9]. Therefore, the overall system security can be improved via identifying and addressing fault-injection and Trojan insertion-based vulnerabilities linked to the extracted FSMs during the pre-silicon design phases [45].

In addition, some FSM-based watermarking strategies embed the authorship information in the states or transitions which need precise FSM extraction to prevent IP infringement [16], [17]. Other FSM-based methods for watermarking alter the *State Transition Graph (STG)* of the FSM subtly for embedding the watermark as a property [16], [17]. Furthermore, partitioned FSM-based sequential logic locking strategies have shown an enormous potential to be more resistant to oracle-guided attacks than combinational logic locking while preventing overproduction [10]–[14]. Researchers have demonstrated that it is essential to understand a design's extracted FSM to reduce the susceptibility to information leakage problems [15]. Additionally, in the hardware verification domain, equivalence checking between the extracted FSMs from higher abstraction levels (such as RTL) and gate-level netlist abstraction should be conducted for secure design transformation and to reduce the verification gap between specification and implementation [18], apart from the security applications mentioned earlier. However, because of the numerous shortcomings of the contemporary gate-level FSM extraction frameworks [26]–[29] as discussed in Section III, many of these hardware protection and validation schemes aimed at increasing the security of an SoC can not be properly implemented in practice, unfortunately. Therefore, a fast, scalable and precise scheme is essential for extracting all the states and transactions of FSMs present in an SoC, particularly for the security-critical IPs.

Although precise extraction of a design's control FSMs is crucial for numerous security and verification applications, the methods and algorithms for FSM extraction reported in the literature primarily focus on extracting FSMs from higher levels of design abstraction (such as RTL) [19], [20]. However, because of design flattening and several optimization stages (e.g., area, power, and performance) performed by the CAD tools, the FSM state registers are mixed with non-FSM

A preliminary version of this work was published in the Proceeding of *2022 IEEE 40th VLSI Test Symposium (VTS)*.

registers during synthesis. As a result, it is challenging to distinguish the FSM state registers and identify the additional don't-care states and don't-care transitions included at the gate-level abstraction succeeding logic synthesis from highly unstructured gate-level netlists. Moreover, identifying all gates in the prospective FSM state registers' feedback loop using the cycle (loop) identification technique [40] exhibits polynomial time complexity. As a result, retrieving every state and transition of a large design's FSMs becomes very difficult. Identifying an FSM from an accumulator or other analogous arithmetic logic blocks with similar feedback loop properties is another challenge in the precise control FSM extraction process. Several recent research works have proposed methods for extracting FSMs from flattened gate-level netlists [26]–[29]. Nevertheless, they are associated with several drawbacks when applied to large-scale and control-intensive benchmarks.

In this paper, we propose a framework named ***F**inite **S**tate **M**achine E**x**tractor Ultra (FSMx-Ultra)* to reconstruct FSMs from synthesized gate-level netlists automatically while taking a short time for computation with 100% accuracy. *FSMx-Ultra* utilizes state-of-the-art efficient graph algorithms and various industry-standard CAD tools based on the proposed mathematical metrics in [29] to recover the control FSMs of designs with diverse sizes and complexity. The *FSMx-Ultra* framework is a rethought version of the recently proposed novel graph theory-based *FSMx* framework [29]. More specifically, our major contributions in this paper are as follows:

- Developing *FSMx-Ultra*, a completely automated framework for fast, scalable, and accurate control FSM extraction from highly unstructured gate-level netlists (either flattened or hierarchical) obtained after logic synthesis;
- Utilizing the state-of-the-art graph theory concepts and the Input Similarity Metric (ISM) and FSM Probability Metric (FPM), presented in [29], to isolate the non-FSM registers with 100% accuracy;
- Extracting human-readable individual gate-level STGs for each of the recognized control FSMs present in the highly unstructured gate-level netlists;
- Demonstrating the efficacy of the *FSMx-Ultra* framework on 14 open-source benchmarks from [21] with different sizes, complexity, and state encoding schemes.

The remainder of the paper is organized as follows. In Section II, we provide an overview and definitions of the terminologies used in the entire paper. Section III discusses the shortcomings of the contemporary FSM extraction techniques and the underlying motivation behind our work. Section IV provides a detailed overview of our proposed *FSMx-Ultra* framework. The experimental results with elaborated algorithmic complexity analysis and efficacy of *FSMx-Ultra* are presented in Section V. Section VI presents the potential applications of the proposed *FSMx-Ultra* framework. Finally, the paper gets concluded with Section VII.

II. Preliminaries and Definitions

Finite State Machine (FSM): From a mathematical standpoint, a *Finite State Machine (FSM)* can be described as a 6-tuple element (S, I, O, s_0, ϕ, λ). Here, S is a finite collection of states, I is a finite set of inputs, O is a finite set of outputs generated from the FSM, s_0 is the reset (or initial) state of the FSM, λ is the output logic function, and $\phi : S \times I \rightarrow S$ is the state transition function that defines the next state of the FSM. In Fig. 1, the generic architecture of a typical FSM is depicted. Three primary components form the high-level architecture of an FSM: (i) the *State Register* (also termed as *State Memory*) storing the current state of the FSM and implementing S, (ii) the combinational *State Transition Logic* implementing ϕ, and (iii) the *Output Logic* of the FSM realizing λ.

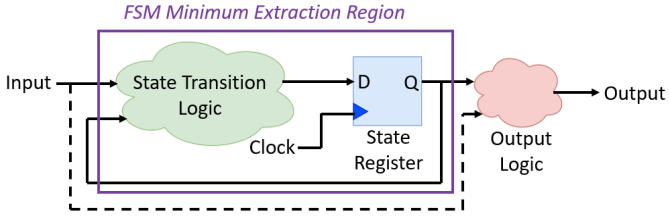


Fig. 1: Architecture of a typical FSM. The black dashed line is present only in the generic architecture of Mealy FSM. The state transition logic and the state register form the minimum extraction region of the FSM.

Moore FSM and Mealy FSM: FSMs can be classified into two major categories considering the type of the output logic: Moore FSM [24], and Mealy FSM [25]. If the output logic of the FSM relies not only on the current state of the FSM but also on the inputs (mathematically $\lambda : S \times I \rightarrow O$), then the FSM is denoted as a Mealy FSM. Conversely, if the output logic of an FSM depends solely on the present state of the FSM, then the FSM is defined as a Moore FSM. The output logic of a Moore FSM can be presented mathematically as $\lambda : S \rightarrow O$. As shown in Fig. 1, the output logic of a Moore FSM is driven only by the FSM state register. However, the output logic of a Mealy FSM is controlled by both the state register and the primary inputs (shown as the dashed line in Fig. 1).
FSM Minimum Extraction Region: The minimum extraction region of an FSM is primarily composed of two parts of the FSM: (i) the *State Register* and (ii) the pure combinational *State Transition Logic*, as defined in the existing literature [26]. The purple-colored bounding box pictured in Fig. 1 represents the minimum extraction region of the FSM. An accurate extraction mechanism of the minimum extraction region of the FSM is required for analyzing the FSM to yield the state transition graph (STG) of the FSM automatically.
Control FSM: When an FSM serves as the control unit of a design, it is termed a control FSM. We provide this definition to distinguish control FSMs from counters. Control FSMs control and sequence operations that take place in the datapath of the design by activating control signals precisely at the required time for action. On the contrary, a counter is typically utilized to count in a pre-defined sequence. For instance, a 3-bit binary counter can generate the count sequence 0, 1, 2, 3, 4, 5, 6, 7, repeatedly providing a specific count value at a particular active edge of the clock signal driving the counter.
State Transition Graph (STG): From mathematical perspective, the *State Transition Graph (STG)* of a control FSM is defined as a directed graph where each node (or vertex) of the graph represents a particular state $s \in S$ and each edge of the graph represents a certain transition between two states, t =

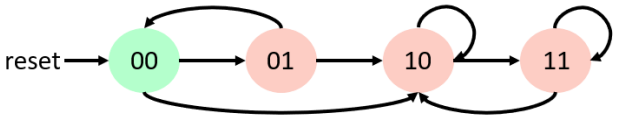


Fig. 2: State transition graph (STG) of a certain control FSM. The nodes and edges of the graph represent the states and transitions between two states of the FSM, respectively.

T (s_i, s_j) from the current state s_i to its next state s_j [45]. The current state s_i and the next state s_j can also be termed as *source state* and *destination state* respectively for the state transition *T* (s_i, s_j). In Fig. 2, the state transition graph of a particular control FSM is depicted. As shown in the figure, the STG has 4 nodes implying the FSM contains 4 states: '00', '01', '10', and '11'. It is also obvious from the figure that there are a total of 8 edges, i.e., state transitions in the FSM. For instance, the state transition *T ('01', '10')* implies that the control FSM switches to the destination state '10' from the source state '01' due to single or multiple transition conditions determined by the state transition logic.

Reset State: The *Reset State* of an FSM is defined as the entry state to the other states existing in the FSM according to FSM automata theory [34]. As the name implies, the reset state of an FSM represents a particular state to which an FSM switches when the reset condition is applied. The reset condition forces the control FSM to transit to the reset state irrespective of the current state of the FSM. For the control FSM depicted in Fig. 2, the '00' state is the reset state.

State Encoding Schemes: States of a particular control FSM can be encoded using three major schemes: *Binary, Gray, and One-Hot*. Using a particular state encoding technique in the design solely relies on the design's optimization goal, such as the design's performance, area, or power consumption. In the binary state encoding scheme, all states of the FSM are enumerated serially initiating from 0 in order of their appearance and will be implemented as a state register having $|\log_2(|S|)|$ numbered flip-flops in hardware, where $|S|$ is the number of states of the FSM. However, in the one-hot state encoding approach, the FSM states are encoded so that all state encoding bits except one are equal to 0 at any point. Consequently, the control FSM state register can be implemented with a register having $|S|$ numbered flip-flops. Finally, in the Gray encoding technique, the states of the FSM are encoded so that the bit difference between the binary represented state encoding values of two consecutive states is 1. The Gray encoding of the states of an FSM results in implementing a state register with a bit width of $|\log_2(|S|)|$ bits, similar to the binary state encoding method. The four states of the control FSM shown in Fig. 2 can be encoded as 2-bit vectors: '00', '01', '10', and '11', respectively, if the binary encoding scheme is applied. The same states can be encoded as '00', '01', '11', and '10' if the Gray encoding technique is employed. Last but not least, if the one-hot encoding approach is utilized, the states need to be encoded with 4-bit vectors: '0001', '0010', '0100', and '1000', respectively.

The power, performance, and area of an FSM are affected by choice of the state encoding scheme. The state encoding choice of the control FSMs influences the overall hardware implementation of a particular design significantly [35]–[37]. The number of state register flip-flops in the binary state encoding scheme is minimal. However, the complexity of the state transition logic increases as the required logic increases. The one-hot encoding scheme increases the number of flip-flops required for the state register. Nevertheless, the one-hot encoding technique simplifies the required logic for the FSM's output and state transition logic. As a result, the combinational circuits present in a typical FSM get more straightforward, reducing the FSM's overall propagation delay. Hence, in turn, the FSM gets compatible with higher clock frequencies, and the timing performance of the design gets improved [35]–[37]. The main drawback of using the one-hot approach is that more flip-flops are required to implement the state register, so the hardware implementation of the FSM requires more area than the other state encoding practices. The Gray encoding approach of the FSM aids in minimizing overall power consumption, reducing the complexity of the state transition logic, and making the asynchronous control signals of the FSM resilient against glitches [35]–[38] while keeping the number of state register flip-flops minimum.

Flattened Netlist and Hierarchical Netlist: The gate-level netlist of a design is a complex interconnection of logic gates via connecting wires. The gate-level netlist of a particular design is obtained using logic synthesis, and the logic gates present in the netlist come from the standard cell library used during the logic synthesis process. The standard cell library contains numerous sequential cells (flip-flops and latches), non-sequential cells (NAND gates, NOR gates, XOR gates, compound gates, tristate buffers, inverters, etc.), and sometimes even macros (memory blocks, IO pads, etc.). The synthesized gate-level netlist of a design is highly unstructured and seems like a sea of logic gates, and the netlist may or may not preserve the modular hierarchy of the design. If the netlist does not preserve any design hierarchy, causes mixing of logic blocks, and allows further optimization by the logic synthesis tool via flattening, then the netlist is termed as *Flattened Netlist*. However, if the designer specifies explicitly not to flatten the synthesized gate-level netlist by the synthesis tool, then the synthesis tool will yield a gate-level netlist preserving the design's modular hierarchy, and mixing logic blocks will not take place. In that scenario, the obtained gate-level netlist after logic synthesis is defined as *Hierarchical Netlist*. *Genus* from *Cadence* and *Design Compiler* from *Synopsys* are two widely used industry-grade logic synthesis tools.

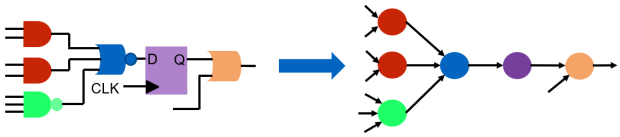


Fig. 3: Directed graph representation of a certain gate-level netlist. The nodes of the graph represent the gates present in the netlist, and the edges of the graph portray the interconnections between two connected gates of the netlist.

Netlist Graph: A synthesized gate-level netlist can be modeled as a complex directed graph from the mathematical viewpoint. The netlist directed graph can be defined mathematically as a 2-tuple entity *G = (V, E)* where *V* is the number

of logic gates i.e. *number of nodes*, and E is the number of interconnections between two connected gates existing in the netlist i.e. *number of edges*. In Fig. 3, the netlist graph representation of a particular gate-level netlist is depicted. Nodes of the graph stand for the logic gates present in the netlist, and edges of the graph imply the interconnections between two connected gates of the gate-level netlist. Graph representation of the netlist makes it suitable for applying existing state-of-the-art efficient graph algorithms to perform topological analysis on the highly unstructured netlist.

III. Relevant Work and Motivation

Precise recognition of FSM structures and isolating the control FSM state registers from the non-FSM registers are challenging in a flattened gate-level netlist due to multi-level optimizations during logic synthesis. The method proposed in [26] was the first extraction scheme of FSMs from a gate-level netlist using topological analysis based on the structural facts of the FSMs to the best of our knowledge. However, this proposed method is associated with several drawbacks, unfortunately. The technique can not successfully isolate control FSMs from accumulators or similar logic blocks since it relies entirely on identifying flip-flops with combinational feedback loops. In addition, the scheme fails to analyze gate-level netlists containing multiple control FSMs and is only applicable to small-sized gate-level netlists.

A strongly connected component (SCC) based control FSM identification methodology was proposed in [27], mounting on [39] to address these limitations. Mathematically, an SCC region is defined as the region of a graph with at least a single cycle (loop). This FSM extraction scheme only aims to identify and analyze the flip-flops having pure combinational feedback loops to detect FSMs present in an SCC region since FSM structures always exist inside SCC regions. Consequently, it fails to isolate control FSM state registers from counter registers since counters have very high structural similarities with control FSMs. Moreover, the approach proposed in [27] did not present any methodology to extract STGs of the recognized FSMs, which does not ensure that the identified registers represent FSMs in practice. Finally, this scheme assumes that control signals generated from the FSMs can be identified in the netlist by examining whether the control signal is connected to the selection pin of a particular multiplexer. However, this method did not clarify how the control output signals can be identifiable in a highly unstructured netlist.

The authors presented an FSM extraction methodology in [28], which considers two structural properties of a control FSM: self and cross flip-flop (FF) influence characteristics. This technique can not perfectly isolate control FSMs from counters despite considering these two structural properties of control FSMs and proposing a scheme for removing counters from the FSM candidates after topological analysis on the gate-level netlist. Furthermore, this approach requires additional manual analysis of the set of final FSM candidates to determine which FSMs are control FSMs. Unfortunately, none of the proposed FSM recognition schemes can extract human-readable gate-level STGs separately for each detected control FSM since these entirely depend on identifying SCCs for FSM region localization. An SCC region may contain multiple FSMs inside, and in that scenario, the proposed scheme will yield a single and composite STG that mingles all the individual STGs of the FSMs. Additionally, the loop identification technique for detecting potential FSM state FFs using [40] exhibits polynomial time complexity and hence has scalability issues even in analyzing medium-sized netlists. Finally, the STG generation technique of the detected FSMs presented in [28] possesses inherent scalability issues since it performs the exhaustive gate-level simulation of the extracted state transition logic of the potential FSM candidates.

Most recently, a novel graph theory-based framework named *FSMx* has been proposed in [29] for fast and precise extraction of all control FSMs present in a flattened gate-level netlist to overcome the mentioned limitations of existing state-of-the-art FSM extraction schemes proposed in [26]–[28]. *FSMx* is much more accurate and roughly 10 times faster on average compared to existing approaches [26]–[28]. However, the framework still suffers from several drawbacks. First of all, the *FSMx* framework can only analyze flattened netlists to extract all control FSMs present. It can not handle gate-level netlists preserving the hierarchy of the design. There are numerous applications where flattening, mixing logic blocks, and further design optimization is strictly prohibited. For example, designers opt for threshold implementation (TI) [42]–[44] to make cryptographic hardware resilient against differential power analysis (DPA) attacks. In that scenario, designers explicitly specify not to flatten the gate-level netlist during logic synthesis, impeding the sharing of logic blocks of the design. As a result, the obtained gate-level netlist conserves the hierarchy of the design, and the *FSMx* framework can not extract control FSMs from such a netlist. Secondly, the framework has inherent scalability issues since it uses exhaustive gate-level simulation to extract STGs of the detected control FSMs. Exhaustive gate-level simulation is entirely prohibited if the number of primary inputs of a design is high or even moderate. Therefore, the framework fails to extract STG of the control FSM if the number of primary inputs of the extracted state transition logic is high or the state transition logic is too complex. Finally, the *FSMx* framework can not extract STGs from benchmarks with a massive number of FSM state FFs. For instance, the memory controller IP core [60] contains a control FSM state register with 66 state FFs, and the framework fails to handle such a scenario. Furthermore, the framework is not scalable to complex benchmarks with a massive number of gates. Hence, there is no guarantee that the proposed framework can extract control FSMs and STGs in every possible use case. To conclude, several issues still need to be resolved for the general adoption of *FSMx* [29].

Our proposed *FSMx-Ultra* framework is an extended version of the *FSMx* framework [29] and intended towards addressing its aforementioned drawbacks. Fast, precise, and automatic extraction of all control FSMs from hierarchical and flattened netlists with their corresponding human-readable gate-level STGs have motivated us in developing *FSMx-Ultra* framework. We primarily focus on extracting control FSMs from synthesized gate-level netlists using standard cell technology libraries. However, the concepts of our proposed *FSMx-Ultra* framework can be easily expanded to support netlists synthesized using *Field Programmable Gate Array (FPGA)* libraries.

IV. FSMx-Ultra Framework

The high-level overview of our proposed *FSMx-Ultra* framework is presented in Fig. 4. From a bird's-eye view, the framework comprises two major modules: the *Netlist Graph Analyzer* module and the *Gate-Level State Transition Graph Extractor* module. The primary purpose of the *Netlist Graph Analyzer* module is to generate the graph representation of the synthesized gate-level netlist and then to perform the topological analysis of the netlist graph based on state-of-the-art efficient graph algorithms. The synthesized gate-level netlist is obtained after logic synthesis of the input *Register-Transfer Level (RTL)* design using any commercial state-of-the-art synthesis tool and can be either flattened or hierarchical. Although we have used *Cadence Genus* as the logic synthesis tool in our experiments, *Design Compiler* from *Synopsys* can also be used. The *Gate-Level State Transition Graph Extractor* module is intended for yielding the individual STGs for each of the recognized control FSMs using *Automatic Test Pattern Generation (ATPG)* techniques. The standard cell technology library in *.lib* format is required during synthesis and is also used by the *Netlist Graph Analyzer* module. Moreover, the standard cell technology library in *.v* format is required by the *TetraMAX* tool from *Synopsys* for generating test patterns. Finally, state encoding information of the control FSMs from the RTL design is needed by the *Gate-Level State Transition Graph Extractor* module to decide how many test pattern files will be generated by the ATPG tool. It is an essential task to extract gate-level STGs of the control FSMs. The state encoding information of a control FSM incorporates the name of the state variable representing a control FSM, the width of the state variable implying the size of the state register, and the state encoding scheme used for the control FSM. Analysis of the synthesis report from *Design Compiler* or the extracted RTL state transition graph of a control FSM by *Cadence JasperGold* provides such essential information.

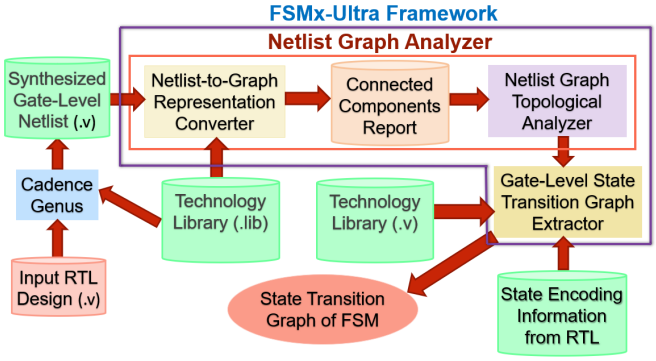


Fig. 4: Overview of the *FSMx-Ultra* framework. The framework analyzes the input synthesized gate-level netlist of a particular RTL design and yields the state transition graphs of the detected control FSMs.

A. Netlist Graph Analyzer

The *Netlist Graph Analyzer* module stands for performing topological analysis based on existing state-of-the-art graph algorithms to identify the portions of the input design netlist representing potential control FSM structures. It can be partitioned into two major sub-modules, as shown in Fig. 4: *Netlist-to-Graph Representation Converter* and *Netlist Graph Topological Analyzer*. The *Connected Components Report* is generated as an intermediary output from the first sub-module, which is eventually taken as input to the second sub-module.

1) Netlist-to-Graph Representation Converter: This sub-module aims to convert the input synthesized gate-level netlist into a directed graph format appropriate for the application of established graph algorithms, as depicted in Fig. 5. The sub-module is composed of three stages: unrolling of the input synthesized gate-level netlist, formation of the intermediate representation of the input unstructured gate-level netlist, and finally, generation of the graph presentation of the input unstructured gate-level netlist.

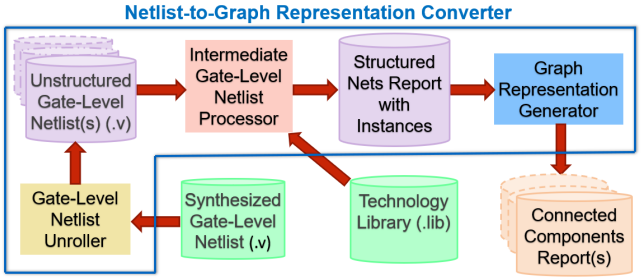


Fig. 5: *Netlist-to-Graph Representation Converter* framework. It generates the associated graph representations of the recognized unstructured gate-level netlists from the input gate-level netlist, which can be either hierarchical or flattened.

Gate-Level Netlist Unrolling: The first stage *Gate-Level Netlist Unroller* takes the synthesized gate-level netlist as input, and the synthesized gate-level netlist can be either hierarchical or flattened. The presence of this netlist unrolling stage is one of the major distinguishing features between the *FSMx* and *FSMx-Ultra* frameworks. *FSMx-Ultra* can analyze hierarchical netlists preserving design hierarchy for this characteristic, which is absent in the *FSMx* framework. If a flattened gate-level netlist of a particular RTL design (having a single module containing the unstructured gate-level netlist of the entire design) is provided as input, the *Gate-Level Netlist Unroller* stage remains inactive. However, in the case of having the hierarchical gate-level netlist of the RTL design (with multiple modules having portions of the entire synthesized netlist), this stage recognizes the design hierarchy, performs unrolling operation, and thus decomposes the input netlist into multiple smaller unstructured gate-level netlists. Each of the obtained unstructured netlists is analyzed individually via the later stages of *Netlist-to-Graph Representation Converter*.

Intermediate Representation of Netlist Formation: The second stage *Intermediate Gate Level Netlist Processor* generates an intermediate representation of the unstructured input gate-level netlist, *Structured Nets Report with Instances*. It is highly structured and acts as the input to the third stage, *Graph Representation Generator*. The intermediate representation of the netlist is used in later stages to reconstruct the fragments of the input netlist representing FSM structures. First, the input and output pins of all the cells in the input technology library are detected. The intermediate representation of the netlist contains the names of the standard cells, cell instance names, pin names, pin types, and the names of the wires connected to the pins in a well-structured manner.

Netlist Graph Representation Generation: The final stage *Graph Representation Generator* analyzes the obtained intermediate representation of the gate-level netlist for constructing the required directed graph of the netlist, *Connected Components Report*. The directed netlist graph can be presented using adjacency list representation. In this representation, a netlist graph is denoted as an entity with numerous pairs of nodes. Each node in a particular pair stands for a cell. Moreover, each pair denotes a particular edge of the netlist graph that implies the link between two interconnected cells since a particular cell's output is connected to another cell's input port. We can find the number of edges of the netlist graph by simply counting the number of pairs. The number of nodes of the netlist graph comes from the total cell count mentioned in the synthesis report generated by the logic synthesizer tool. The adjacency list format of the netlist graph has been chosen to minimize the complexity of the employed graph algorithms. For a flattened netlist, a single, gigantic and complex netlist graph representation is generated by this sub-module. Nonetheless, multiple relatively simpler and smaller netlist graph representations are yielded by this sub-module if a hierarchical netlist is analyzed due to the gate-level netlist unrolling stage present. Analysis of multiple simpler and smaller netlist graphs individually comes with more inherent computational advantages than analysis of a single but highly complex and gigantic netlist graph. Consequently, the *FSMx-Ultra* framework extracts control FSMs from the hierarchical netlist of a particular RTL design more quickly than the flattened netlist of the same RTL design. We have demonstrated this fact of the *FSMx-Ultra* framework in Section V-D.

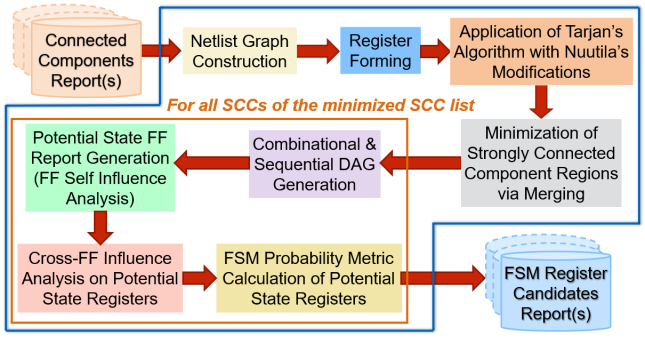
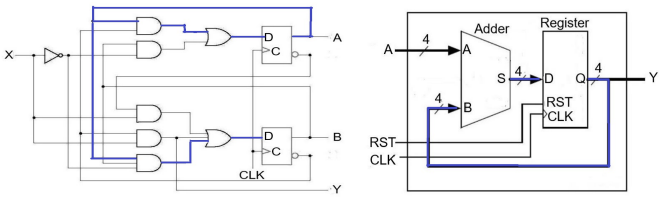


Fig. 6: *Netlist Graph Topological Analyzer* framework. It generates a list of FSM register candidates with the maximum values of *FPM*. Later stages use these register candidates to reconstruct the control FSM netlists.

2) Netlist Graph Topological Analyzer: This sub-module is intended for performing graph algorithmic analysis on the graph representation of the netlist *Connected Components Report* obtained from the previous sub-module. The high-level overview of the *Netlist Graph Topological Analyzer* framework is presented in Fig. 6. Analyzing the input graph representation report generated by the previous sub-module, this framework first constructs the *Netlist Graph*, which is often highly complex with a large number of nodes and edges in case of practical benchmarks, as illustrated in Table I. Next, all the flip-flops are identified in this netlist graph, and registers are formed by grouping them. Then, the *Tarjan's Strongly Connected Components Algorithm with Nuutila's Modifications* [41] algorithm is applied on the obtained *Netlist Graph*. The main reason behind choosing this algorithm is its memory efficiency while keeping similar time complexity to the proposed Tarjan's algorithm for finding SCC regions [39].

This stage decomposes the whole netlist graph into smaller sub-graphs representing the graph's strongly connected component (SCC) regions. We are only interested in analyzing the SCC regions of the graph since these regions contain potential FSM structures mathematically. This process resembles the *divide-and-conquer* approach and makes a clear distinction between *FSMx-Ultra* and *FSMx* proposed in [29]. Performing analysis on smaller sub-graphs has more computational advantages than analyzing the entire giant graph, especially for larger netlist graphs. It is one of the underlying reasons that explain why *FSMx-Ultra* is so much faster compared to *FSMx*, which is also evident from the experimental results shown in Table I. Detailed algorithmic complexity analysis from the mathematical viewpoint is presented in Section V-A.

In the next stage, the number of SCC regions is minimized. A single flip-flop, a part of a particular register, with a combinational feedback loop can also form a separate SCC region while other flip-flops of that register exist in another SCC region. Hence, these SCC regions can be merged to form a single SCC region instead of two (the *modified SCC*). In this way, a list of minimized SCC regions is constructed. This minimization process also helps to improve overall run-time. Finally, structural analysis (marked by the orange bounding box in Fig. 6) is performed on each of the SCC regions present in the minimized SCC region list using the novel graph theory-based approach, and mathematical metrics presented in [29] and a list of FSM register candidates is obtained as the output from this sub-module.



(a) FSM of a sequence detector [30] (b) A 4-bit accumulator [31]

Fig. 7: An FSM and accumulator example [29].

The structural analysis stage on the modified SCC regions is crucial for precisely identifying the control FSM register candidates. The central point to be noted here is that this sort of analysis is performed on sub-graphs representing SCC regions by *FSMx-Ultra*. However, *FSMx* performs this analysis on the entire netlist graph. Hence, its overall time complexity is higher than *FSMx-Ultra*. The structural truths of accumulators, data registers, control FSMs, and counters were thoroughly investigated in [29], and three essential properties were found based on the implementations of these entities. Those properties (P) were used to derive and formulate two important mathematical metrics for isolating FSM registers from the non-FSM ones, and *FSMx* did not require any further post-processing stage or human decision [29].

The first property (P-I) states that *data (D) inputs of potential state FFs are driven by dissimilar standard cells* [29]. This property can effectively separate data registers

from counters, accumulators, and control FSMs. The data registers form the essential part of the data flow in a design. Therefore, similar standard cells tend to drive the D-inputs of flip-flops, constituting a data register after logic synthesis [32]. Counter and accumulator register FFs also exhibit this property occasionally, which is apparent from Fig. 7b. On the contrary, dissimilar standard cells tend to drive the D-inputs of the control FSM FFs since those represent the control flow of a design [32]. It is also evident from Fig. 7a. P-I was utilized to develop a metric called *Input Similarity Metric (ISM)* in [29] to calculate the D-input similarity of the FFs present in a particular register which was denoted as follows:

$$ISM = \frac{max(N_1, N_2, N_3, ...)}{N} \times 100\% \quad (1)$$

Here, *N* represents the size of a certain register which implies that it consists of a total of N FFs. Among the N FFs, N_1 FFs have one type, N_2 FFs have another type of cells driving their corresponding D-inputs, and it goes on similarly. The maximum of all these values was taken since we want to consider the maximum similarity in the worst-case scenario. As depicted in Fig. 7a, the control FSM has an ISM of 50%. It is because a 2-input OR gate drives the D-input of a FF of the FSM register. Additionally, the D-input of the other FF gets driven by a 3-input OR gate. This scenario makes to have max(N_1, N_2) = 1 and N = 2 (as the control FSM consists of 2 FFs). Conversely, as illustrated in Fig. 7b, the 4-bit accumulator has an ISM of 100%. The underlying reason behind this is the presence of four 2-input XOR gates in the adder block, which drive the four FFs of the accumulator register. It means that a single type of standard cell is driving all four FFs. In other words, we get N = max(N_1) = 4. We have set ISM = 85% as the threshold *R* for eliminating the non-FSM registers similar to it was done for *FSMx* [29].

The second property (P-II) is presented as *potential state FFs must contain pure combinational self-feedback loops* in [29]. It implies that each FF of a particular FSM register should influence itself via at least one combinational feedback loop. From the graph theory perspective, the same FF should be reachable through a combinational logic starting from a particular FF. It can also be observed in Fig. 7a. Mathematically, it gives birth to a parameter named *Self-influence Parameter* and a register with *N* FFs must have a self-influence parameter of N [29]. However, this property also exists in the accumulator structure [27], which is evident for the 4-bit accumulator example presented in Fig. 7. Hence, a clear distinction is required between FSM and accumulator structures. The third property (P-III) was presented to accomplish such an objective and narrated as *potential state FFs of a prospective FSM register should influence the rest of the state FFs and must also be influenced by the other state FFs of that register* [29]. This property is absent in accumulator structures and emphasizes the cross-influence characteristics of control FSM structures which ultimately helps in finding another parameter called *Cross-influence Parameter*. A potential FSM state register of size *N* should have N(N – 1) as the value of this parameter. P-II and P-III were combined to develop the second mathematical metric named *FSM Probability Metric (FPM)* in [29] which calculates the probability of a register present in the SCC region of being an FSM. FPM was defined as follows:

$$FPM = \frac{S + C}{N^2} \times 100\% \quad (2)$$

Here, S is the number of self-influence paths, and C represents the number of cross-influence paths. Finally, N stands for the size of the register. For the FSM of the sequence detector, shown in Fig. 7a, we get N = 2, S = 2 and C = 2. Hence, it exhibits an FPM of 100%. On the other hand, the 4-bit accumulator shown in Fig. 7b has an FPM of only 25% since for it N = 4, S = 4 and C = 0. This noticeable difference between the FPM values of the accumulator and the control FSM can be used to remove accumulator structures.

In the structural analysis stage of the modified SCC regions after the minimization process, we deconstruct a particular modified SCC region into two directed acyclic graphs (DAGs), namely *Combinational DAG and Sequential DAG*. Since these DAG portions do not contain any cycle (loop) inside, analysis of those provides tremendous computational advantages inherently [29]. Analysis of a cyclic graph directly is computationally more expensive. The sequential DAG contains all the edges of the SCC region sub-graph, with one node of the edge representing a sequential cell (FF or latch) and the other one standing for a non-sequential cell. Conversely, the rest of the edges of that modified SCC region is accommodated by the combinational DAG. Let us assume that the modified SCC can be represented as a sub-graph, *G = (V, E)*. Hence, *E* gets minimized to E_c holding only the edges between two non-sequential cells, analytically. The remaining portion of *E* belongs to the sequential DAG. It contains only the edges between a sequential cell and a non-sequential cell. Moreover, *V* gets partitioned into two parts. The first part, V_s, holds all the sequential cells (i.e., flip-flops and latches). The remaining nodes V_c holding the rest of the non-sequential cells form the second portion. Next, ISM is calculated based on Eq. 1, and the sequential DAG is minimized mounting on the obtained ISM value. All the registers with ISM exceeding the threshold *R* are discarded by *FSMx-Ultra*. We term the registers remaining in the minimized form of the sequential DAG as *Potential State Registers*. In different words, utilizing the ISM, potential state FF vertices V_r are extracted from V_s via minimization. Logically, V_r is only a small fraction of V_s in number.

Lastly, the starting and ending points of the FFs of the potential state registers are detected. Then, we apply *Depth-First Search (DFS)* on the combinational DAG to analyze self-influence and cross-influence among the FFs of a particular register. Moreover, registers having V_r are analyzed instead of considering the entire V_s. These two actions help to improve the overall run-time of *FSMx-Ultra*. Additionally, those simplify post-processing methods for precise extraction of the control *FSM Netlists* [29]. We adopted the same FPM-based post-processing method as *FSMx*. The *FSM Register Candidates Report* contains all the names and sizes of the potential state registers. Moreover, the obtained FPM and the extracted FSM candidates region (having FF names and other gates for a potential FSM register) are also included. If a flattened gate-level netlist is analyzed by *FSMx-Ultra*, then a single *FSM Register Candidates Report* is generated.

Nonetheless, in the case of analyzing hierarchical gate-level netlists, such a report is generated multiple times due to the unrolling stage presented in Section IV-A. This report is an important input for the subsequent stages of *FSMx-Ultra*.

B. Gate-Level State Transition Graph Extractor

The primary objective of this module is the automatic extraction of the associated state transition graphs (STGs) of the recognized control FSMs. The extracted STGs by our proposed *FSMx-Ultra* framework are human-readable and identical to the STGs generated by the recently proposed *FSMx* framework [29]. Additionally, the *Gate-Level State Transition Graph Extractor* module of the *FSMx-Ultra* framework seems to be functionally analogous to the *Gate-Level Boolean Function Analyzer* module of the *FSMx* framework. However, two major differences between these modules make the *FSMx-Ultra* framework unique in terms of performance and scalability.

The *Gate-Level State Transition Graph Extractor* module takes the state encoding information of the control FSMs present in the RTL description of the design as an additional input which is absent in the *Gate-Level Boolean Function Analyzer* module of the *FSMx* framework. This input contains the name of the FSM state variable and its width with the utilized state encoding scheme, which can be readily obtained from existing commercial synthesis or formal verification tools like *Design Compiler* or *Cadence JasperGold* as mentioned before. This additional input's primary purpose is the partial contribution to making the proposed *FSMx-Ultra* framework more scalable compared to the *FSMx* framework by aiding in determining the number of test pattern files generated by the ATPG tool from *Synopsys* named *TetraMAX*.

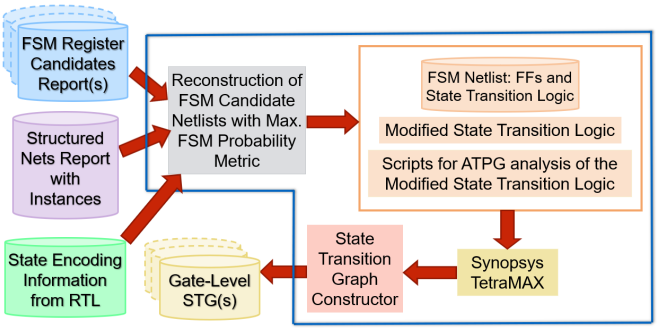


Fig. 8: Framework of the *Gate-Level State Transition Graph Extractor* module. It performs ATPG-based analysis to extract the gate-level STGs of the detected control FSMs.

Furthermore, the *Gate-Level State Transition Graph Extractor* module performs ATPG-based analysis to extract the gate-level STGs of the control FSMs. On the contrary, the *Gate-Level Boolean Function Analyzer* module of the *FSMx* framework performs the exhaustive gate-level simulation of the extracted pure combinational state transition logic using the corresponding automatically generated *Verilog* testbenches. Exhaustive gate-level simulation fails if the state transition logic of a particular FSM is highly complex or the number of primary inputs of the state transition logic is high or even moderate. Therefore, this module of the *FSMx* framework [29] fails to handle such possible use cases and extract gate-level STGs of the identified control FSMs, suffers from inherent scalability issues, and is not applicable for analyzing any flattened gate-level netlist in general. The high-level overview of the framework of the *Gate-Level State Transition Graph Extractor* module is shown in Fig. 8. The operation of this module can be decomposed into three major stages: reconstruction of the control *FSM netlist*s, generation of the *Modified State Transition Logic*, and extraction of the individual gate-level STGs of the identified control FSMs.

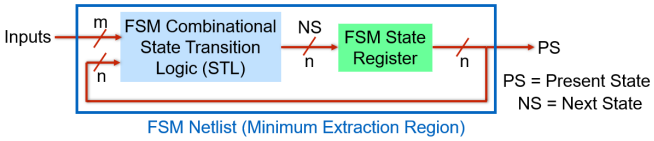


Fig. 9: Generic architecture of the *FSM Netlist*. 'm' and 'n' represent the bus sizes of the associated inputs and outputs. The FSM state register is 'n'-bit wide and has 'n' flip-flops.

Reconstruction of FSM Netlist: The *Gate-Level State Transition Graph Extractor* module takes the previously generated *FSM Register Candidates Report* and *Structured Connected Components Report* as its major inputs. FSM candidate netlists having the maximum *FSM Probability Metric* are reconstructed utilizing these two inputs. One major output after such a process is the automatic extraction of all such *FSM Netlist*s. A particular control *FSM Netlist* is composed of only the flip-flops and pure combinational state transition logic. Hence, it serves as the *Minimum Extraction Region* of a certain control FSM [26] as shown in Fig. 1. A more detailed view of the *FSM Netlist* architecture is depicted in Fig. 9. The data *D* inputs to the flip-flops forming the FSM state register are termed as *Next State (NS)* and the data *Q* outputs from those flip-flops are called *Present State (PS)* collectively. The *Inputs* refer to the primary inputs of the *FSM Netlist*. The *Next State* of the FSM is solely determined by the *Present State* and the *Inputs* and can be represented mathematically as $NS = f(Inputs, PS)$. The extracted pure combinational state transition logic of the *FSM Netlist* implements the state transition function f and serves as an essential entity for yielding the gate-level STG of the detected control FSM.

Generation of Modified State Transition Logic: Another major output from the *Gate-Level State Transition Graph Extractor* module is the associated *Modified State Transition Logic* of the *FSM Netlist* after the reconstruction phase is over. The pure combinational state transition logic of the *FSM Netlist* is modified to make it suitable for performing ATPG-based simulation and analysis to assist the automated extraction process of its corresponding gate-level state transition graph. ATPG-based analysis feature of *FSMx-Ultra* makes the exhaustive gate-level simulation of the extracted state transition logic entirely obsolete performed by the *FSMx* framework. Hence, our proposed *FSMx-Ultra* framework resolves the inherent scalability issues of the *FSMx* framework [29]. Finally, the *State Encoding Information from RTL* is also required to determine the number of test pattern files to be yielded by the *Synopsys TetraMAX* tool while generating *Tcl* scripts for performing the ATPG-based analysis of the extracted *Modified State Transition Logic*.

The generic architecture of the *Modified State Transition*

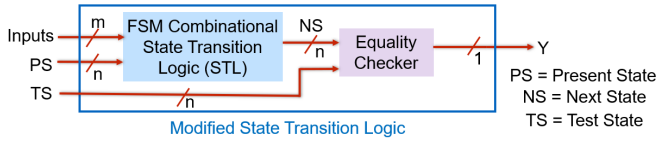


Fig. 10: General architecture of the *Modified State Transition Logic*. 'm' and 'n' represent the bus widths of the associated inputs and intermediate outputs. The final output 'Y' is a single wire which provides the equality checking result.

Logic is shown in Fig. 10. It is primarily composed of two blocks: the pure combinational state transition logic of the control FSM and an equality checker circuit connected to its output. The pure combinational state transition logic of the control FSM can be readily extracted when the *FSM Netlist* gets reconstructed as it is an integral part of the control *FSM Netlist*. As mentioned before, the *Next State (NS)* is a direct function of the *Present State (PS)* and the primary *Inputs* of the control FSM. The equality checker block checks whether the *Test State (TS)* matches with the *Next State (NS)* or not. It generates '0' at the output 'Y' if *TS* matches with the *NS* otherwise '1' is generated at 'Y'. Therefore, the 'n' bits of *NS* can be logically XORed with the 'n' bits of *TS*, and the outputs of XOR gates can be ORed together. This logical configuration represents the implementation of the equality checker from a high-level perspective. Our implementation of the equality checker uses only 2-input XOR gates, 2-input OR gates, and interconnections between them.

We need to provide all possible logical values of *TS* as an input of the equality checker. These possible logical values of the next states should be finite as those depend on the FSM encoding style. Information on the control FSM encoding style can only be obtained from the design's RTL description. It is impossible to get such a piece of important information after performing the logic synthesis of a design. Hence, the *State Encoding Information from RTL* is required as a major input to our proposed *FSMx-Ultra* framework. If the control FSM is encoded using *Binary* or *Gray* encoding scheme, 2^n combinations of the logical values are applied at the *TS* sequentially one at a time and checked for matching with the value at *NS*. On the other hand, if the control FSM is encoded using the *One-Hot* technique, the 'n' combinations of test values are checked sequentially, as mentioned, keeping only a single bit active (set to '1') at a time. In this manner, the *State Encoding Information from RTL* determines how many times the ATPG tool named *Synopsys TetraMAX* should run, and thus help to make the *FSMx-Ultra* framework scalable by keeping the overall run-time limited.

Extraction of the Gate-Level STG: The most interesting processing phases of the *Gate-Level State Transition Graph Extractor* module start from when *Tcl* scripts are generated automatically for performing ATPG-based simulation and analysis of the *Modified State Transition Logic* using the ATPG tool named *Synopsys TetraMAX*. These scripts for running the ATPG tool are planned for generating test patterns that violate the *stuck-at-1 (SA1)* condition at the output wire 'Y' of the *Modified State Transition Logic*, equivalent to removing all faults and generating test patterns for *SA1* fault at 'Y' sequentially for all possible combinations determined in the previous stage. Therefore, the ATPG tool must yield '0' at 'Y' to generate test patterns for this fault. It implies that *TS* has matched perfectly with *NS*.

In addition, we have used *'n-detect'* option of *Synopsys TetraMAX* to generate 200 test patterns for such a perfect match. As a result, 2^n test pattern files are generated sequentially in total if the control FSM is encoded with *Binary* or *Gray* encoding technique, else 'n' numbered test pattern files are produced. Each test pattern file contains 200 test patterns for the *SA1* fault at 'Y' if it is not empty. Present state and next state information, which is crucial for generating the gate-level STG of the control FSM, can be extracted after rigorous analysis of the obtained test patterns. Empty test pattern files stand for the unmatched scenarios of *TS* and *NS*, implying such state transitions are not possible. The *State Transition Graph Constructor*, as depicted in Fig. 8, implements this processing stage which eventually extracts the gate-level STGs of the control FSMs present in the netlist in both textual and graphical representations.

Finally, we have used the *PyGraphviz* package to yield the gate-level STG in graphical representation. Moreover, the conditions for a particular state transition between two states can be found via analyzing the obtained test patterns from the *Synopsys TetraMAX* tool. Such conditions are also reported in the textual presentation of the gate-level STG and will assist designers in performing security assessments in later stages, such as fault-injection and information leakage assessments. Additionally, this information can aid designers in developing novel FSM-based watermarking and sequential logic locking schemes. Last but not least, we have compared the extracted gate-level STGs of the control FSMs of the open-source benchmark designs, enlisted in [29], by the *FSMx* framework with the STGs generated by the proposed *FSMx-Ultra* framework. We have found that all of the extracted gate-level STGs by these two frameworks are identical, which suggests that the generation of 200 test patterns is quite enough and effective for obtaining the entire gate-level STGs of the associated control FSMs by our proposed *FSMx-Ultra* framework.

V. Experimental Results and Discussion

A. Algorithmic Complexity

1) Time Complexity: Our proposed *FSMx-Ultra* framework utilizes the *divide-and-conquer* strategy to decompose the entire graph of input gate-level netlist, which is often quite massive with a large number of nodes and edges, into smaller sub-graphs by applying the *Tarjan's Strongly Connected Components Algorithm with Nuutila's Modifications* [41]. This efficient graph algorithm is an improved version of the *Tarjan's Strongly Connected Components Algorithm* [39] which can identify all the sub-graphs of the input graph having at least one cycle (loop) inside and can be applied to directed graphs. The time complexity of the algorithm presented in [41] is $O(|V| + |E|)$ where $|V|$ is the number of vertices (nodes) and $|E|$ is the number of edges of the graph. Due to the associated linear time complexity, identifying the graph's strongly connected components (SCCs) is rapid.

The *FSMx-Ultra* framework analyzes the detected SCCs further, as discussed in Section IV-A for precise recognition of the control FSM structures present in the synthesized gate-level netlist. Let us assume that 'k' is the total number of

TABLE I: Worst-case run-time comparison between *FSMx* and *FSMx-Ultra* for 14 benchmarks obtained from [21]–[23].

Benchmark Name	Gate Count	FF Count	Edge Count	Control FSM Count	Control FSM-FF Count	Control FSM ISM(%)	*FSMx* Run-Time	*FSMx-Ultra* Run-time
UART Core [49]	576	89	1402	2	3, 3	67, 33	0.8 s.	0.4 s.
XTEA Cipher [50]	955	105	2214	1	2	50	1 s.	0.5 s.
SAYEH CPU [51]	1320	170	9601	1	4	50	10 s.	6 s.
CMAC Cipher [52]	1549	264	3255	1	3	67	1 s.	0.6 s.
SHA-256 [53]	5254	1806	120075	1	2	50	19 s.	2.5 s.
SHA-512 [54]	10763	3666	361582	1	2	50	3 min. 28 s.	10 s.
POLY1305 MAC [55]	11586	1724	81709	2	3, 3	67, 33	4 min. 30 s.	18 s.
AES-128 [56]	12976	2987	838121	4	2, 2, 2, 2	50, 50, 50, 50	6 min. 20 s.	31 s.
Tiny MIPS CPU [57]	17443	9285	662773	1	4	50	4 min. 11 s.	57 s.
Smart Card RSA [58]	35521	14578	83882	3	2, 3, 4	50, 67, 75	2 min. 16 s.	39 s.
USB HOST [59]	3163	1326	9524	1	4	50	N/A	6 s.
Memory Controller [60]	3207	1051	8489	1	66	50	N/A	8 min. 45 s.
PicoRV32 CPU [61]	6439	1680	19278	3	3, 2, 2	33, 50, 50	N/A	10 s.
OpenRISC 1200 [62]	201445	69943	492286	5	2, 2, 2, 3, 3	50, 50, 50, 67, 33	N/A	6 h. 55 min.

modified SCCs after the SCC merging phase (if required) and focus only on a modified SCC for detailed analysis. A modified SCC is deconstructed by *FSMx-Ultra* into two acyclic sub-graphs called *Sequential DAG and Combinational DAG*. The *Sequential DAG* can be represented as a 2-tuple entity G_r = *(*V_r*,* E_r*)*, where V_r and E_r represent its nodes and edges, respectively. Similarly, The *Combinational DAG* can be represented as a 2-tuple entity G_c = *(*V_c*,* E_c*)*, where V_c and E_c represent its nodes and edges, respectively. The overall time complexity for analyzing a single modified SCC is $O(|V_r| \times (|V_c| + |E_c|))$ which can be derived similarly as presented in [29]. We need to do this sort of analysis for 'k' numbered modified SCCs. Therefore, the overall time complexity for this stage is $O(\sum_{i=1}^{k}[|V_{ri}| \times (|V_{ci}| + |E_{ci}|)])$.

Since this analysis stage is associated with quadratic time complexity, it has the dominant effect on the overall run-time of the *FSMx-Ultra* framework. The time required for detecting SCCs is minimal compared to this, therefore having a minor effect on the overall run-time of *FSMx-Ultra* and can be neglected. Moreover, the overall time required by the ATPG tool to aid in extracting gate-level STGs is so small that it is also negligible. Furthermore, the other processing phases require reading from and writing into text files, which can also be ignored. An important fact of the *FSMx-Ultra* framework is that it analyzes the detected SCCs, which are small portions of the entire netlist graph. As a result, this *divide-and-conquer*-based processing phase has inherent computational advantages over the analysis feature of *FSMx* [29]. On the other hand, *FSMx* performs analysis on the entire netlist graph, which is often extremely complex in practical benchmarks [21]–[23]. Additionally, the exhaustive gate-level simulation of the extracted state transition logic to yield gate-level STGs also adversely affects the performance and scalability of *FSMx*. Hence, *FSMx* is much slower and less scalable compared to *FSMx-Ultra* when both of them analyze a large and complex netlist graph. It is also evident from the experimental results presented in Table I. From this mathematical analysis, we can easily make a logical conclusion that *FSMx-Ultra* is much faster compared to the existing methods [26]–[28] since *FSMx* is 10 times faster on average than those approaches [29].

2) Space (Memory) Complexity: In the worst-case scenario, the entire input netlist graph can form an SCC region. Hence, the input netlist graph must be stored in the computer memory stack. Therefore, the space complexity of *FSMx-Ultra* is approximately *O(V)* since the entire netlist graph containing all the nodes must be stored in the computer memory stack. It is roughly equal to the space complexity of the modified and improved version of Tarjan's SCC algorithm proposed in [41]. The space complexity of *FSMx-Ultra* is comparable to existing SCC-based FSM recognition approaches [27], [28]. However, as presented in [29], the space complexity of *FSMx* is $O(|V_r| + |V_c|)$, where $|V_r|$ and $|V_c|$ are the number of nodes of the sequential and combinational DAGs of the netlist graph, respectively. It is smaller than the overall space complexity of *FSMx-Ultra*. From this analysis, it gets evident that *FSMx-Ultra* requires only a bit more memory compared to *FSMx*. Nonetheless, *FSMx-Ultra* supersedes *FSMx* in terms of run-time, performance and scalability. Hence, the proposed *FSMx-Ultra* framework is a more promising solution compared to the state-of-the-art FSM extraction methodologies [26]–[29].

B. FSM Extraction Run-time

The *FSMx-Ultra* framework was implemented using *Python* programming language to develop an automated tool. We have used the *NetworkX* [48] package to apply the efficient graph algorithms for analyzing the netlist graph as discussed in Section IV. The package contains almost all existing state-of-the-art graph algorithms. We have used *Cadence Genus* as the logic synthesizer to obtain the flattened gate-level netlists of the 14 benchmarks shown in Table I. The typical version of the standard cell technology library *Synopsys SAED90nm* in *.lib* format was used during synthesis. However, *FSMx-Ultra* does not restrict the application of other available standard cell technology libraries for academic and industry usage. We have examined the effectiveness of the automated tool implementing our proposed *FSMx-Ultra* framework via analyzing synthesized netlists which used 10 different industry-standard technology libraries from *Cadence*, *Synopsys* and *GlobalFoundries* during synthesis. It was found that *FSMx-Ultra* supported netlists synthesized using all the 10 standard cell technology libraries under test. However, the tool implementing the FSM extraction framework proposed in [28] supports fewer standard cell technology libraries till now, to the best of our knowledge. It implies that *FSMx-Ultra* is more efficacious than that. *FSMx* also supports several standard cell technology libraries [29].

Since the *FSMx* framework is faster compared to the previously proposed approaches [26]–[28], we have compared the performance of our proposed *FSMx-Ultra* framework with it. The worst-case run-time comparison between these two frameworks has been illustrated in Table I. The run-time of *FSMx* and all other experimental data except the run-time of *FSMx-Ultra* was obtained from [29]. The name of the

TABLE II: Worst-case run-time of FSMx-Ultra for Memory Controller [60] with different netlist type and state encoding.

Netlist Type	State Encoding	Gate Count	FF Count	Edge Count	Control FSM-FF Count	Control FSM ISM(%)	Run-time
Flattened	Binary	3137	992	8363	7	57	43 s.
	Gray	3185	1008	8412	7	57	54 s.
	One-Hot	3207	1051	8489	66	50	8 min. 45 s.
Hierarchical	Binary	3292	1006	9245	7	57	31 s.
	Gray	3248	1032	9378	7	57	42 s.
	One-Hot	3322	1065	9536	66	50	6 min. 35 s.

benchmarks, gate count, flip-flop count, edge count, control FSM count, control FSM-FF count, ISM (in %) of the control FSM, and overall run-times of the *FSMx* and *FSMx-Ultra* frameworks have been presented. The FPM (in %) of all the detected control FSMs was found to be 100%. We set ISM of 85% as the threshold R to remove the non-control FSM registers to provide more flexibility, similar to *FSMx* [29]. We performed all the experiments on the flattened netlists of the benchmarks using an Intel Core i7-1065G7 processor clocked at 1.3 GHz with 16GB RAM on a personal desktop. Both frameworks analyzed the flattened gate-level netlists of the open-source benchmarks collected from [21]–[23]. From the last 4 rows of Table I, it is evident that *FSMx* fails to handle flattened netlists of more complex and larger benchmarks. *FSMx* was unable to extract the gate-level STGs of the control FSM of the *Memory Controller IP* [60] since the FSM has a state transition logic with a massive number of primary inputs. As a result, the *Gate-Level Boolean Function Analyzer* of *FSMx* failed due to inherent scalability issues since it tried to perform the exhaustive gate-level simulation of the extracted combinational state transition logic. The same thing is true for the *OpenRISC 1200* CPU [62], which is a gigantic netlist graph with an enormous number of nodes and edges and other benchmarks presented in Table I. The run-times of *FSMx* have been denoted as *Not Available (N/A)* in such scenarios since obtaining the overall run-time was practically infeasible. The FSM extraction schemes presented in [26], [28] are also associated with similar scalability issues and fail to analyze complex netlist graphs. However, all control FSMs of the netlists were extracted by *FSMx-Ultra* for all the benchmarks presented much faster, as shown in Table I. It suggests that *FSMx-Ultra* is better than existing methods in terms of performance, run-time, and scalability.

C. FSM Extraction Accuracy

The gate count and flip-flop count, as shown in Table I, were obtained from the *Cadence Genus* generated synthesis report. The edge count was obtained from the report generated by *Netlist-to-Graph Representation Converter*. The total count of the control FSM and the corresponding FFs were obtained from the benchmarks' RTL descriptions. An industry-grade formal verification tool *Cadence JasperGold* was used for this purpose, along with extracting the RTL STG of the control FSMs of the benchmarks. The ISM (in %) of the control FSMs were obtained from the reports generated by both *FSMx* and *FSMx-Ultra*, and those were identical. Since *FSMx* is more precise than other approaches [26]–[28] and can identify hidden don't-care states and transitions in the netlist abstraction, we have compared its extracted gate-level STGs with the ones yielded by *FSMx-Ultra* for the first 10 benchmarks presented in Table I in which analysis performed by *FSMx* was successful. It was observed that all the gate-level STGs obtained by these two frameworks matched perfectly. It must be noted that *FSMx* was able to extract the control FSM netlists for all the benchmarks but failed to extract the gate-level STGs for the last 4 benchmarks due to scalability issues, as mentioned earlier. Moreover, we also compared the RTL STGs extracted by *Cadence JasperGold* with the gate-level STGs obtained by *FSMx-Ultra* as it was performed in [29]. We found that *the RTL STG of a control FSM is a subset of its gate-level STG*. Finally, we have used *Synopsys Formality*, a formal verification tool, to compare the extracted control FSM netlists by the *FSMx* and *FSMx-Ultra* frameworks. It was noted that the extracted FSM netlists matched properly. We have also compared the extracted control FSM netlists with their corresponding RTL descriptions using the same tool, and perfect matching was obtained. To conclude, all these employed validation methods suggest that the accuracy of the *FSMx-Ultra* framework is 100%, even when *FSMx* failed to extract the gate-level STGs. It makes *FSMx-Ultra* a more accurate solution compared to state-of-the-art methods [26]–[29] to extract control FSMs from synthesized netlists.

D. Case Studies

We have presented case studies on two practical benchmarks from [21] to demonstrate that our proposed *FSMx-Ultra* framework can extract FSMs from complex and large benchmarks, although the recently proposed *FSMx* framework failed, as shown in Table I. The first benchmark is the *Memory Controller* IP and the second is the *OpenRISC 1200* CPU.

1) Memory Controller: The *Memory Controller* from [21] is intended for various embedded applications. It supports SDRAM, SSRAM, FLASH memory, ROM, and several other devices. It has eight chip selects, and each of them is programmable. Moreover, it provides default boot sequence support with other important features [60]. The IP has a single control FSM, as evident from Table I. We have analyzed the flattened gate-level netlists of this benchmark with 3 different state encoding schemes, namely *Binary, Gray, and One-Hot*, using *FSMx-Ultra* to illustrate that our proposed framework can extract control FSMs utilizing the conventional state encoding practices. Moreover, we have also performed analyses on the hierarchical netlist of this design with the mentioned 3 state encoding schemes. The obtained experimental results are presented in Table II. *FSMx* extracted the control FSM netlist but failed to yield the gate-level STG in all the mentioned scenarios in this table. However, *FSMx-Ultra* succeeded in handling all such use cases, as evident from Table II. It points to the general applicability of *FSMx-Ultra* in analyzing flattened and hierarchical netlists having control FSMs utilizing various state encoding approaches. The extracted gate-level STG of the control FSM of the *Memory Controller* using

Binary state encoding scheme is depicted in Fig. 11. As evident from the figure, the gate-level STG is quite complex, which was extracted using ATPG-based analysis by *FSMx-Ultra*. The extracted gate-level STG contains the 66 states present in the RTL description with hidden don't-care states and transitions. Exhaustive gate-level simulation performed by *FSMx* to extract this gate-level STG fails since the extracted combinational state transition logic contains 98 primary inputs, and testing 2^{98} patterns is practically infeasible.

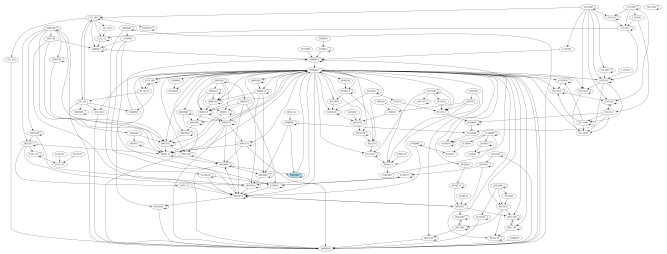


Fig. 11: The extracted gate-level STG of the *Memory Controller* IP [60] with binary state encoding scheme employed. The control FSM has 98 primary inputs in the pure combinational state transition logic. Moreover, the gate-level STG of the control FSM is quite complex, with the 66 states present in the RTL description and hidden states and transitions.

2) OpenRISC 1200: The *OpenRISC 1200* CPU from [21] is a 32-bit scalar RISC utilizing Harvard micro-architecture and 5-stage integer pipeline with virtual memory support (MMU) and basic DSP capabilities. It is an implementation of the OpenRISC 1000 processor family. Additional features incorporate a high-resolution tick timer, programmable interrupt controller, debug unit for real-time debugging purposes, and power management support [62]. Analysis of the flattened gate-level netlist of this processor core was the most challenging among all the benchmarks presented in Table I since it contains 201,445 gates (nodes) with 69,943 flip-flops and 492,286 interconnections between two gates (edges). Unfortunately, none of the existing FSM extraction frameworks [26], [28], [29] were validated on such a huge and complex benchmark. We tried to analyze this huge netlist graph with those methods. It is quite unfortunate that all of those failed due to their inherent scalability issues since this CPU core contains control FSMs with a complex pure combinational state transition logic with a massive number of gates and primary inputs. Nevertheless, *FSMx-Ultra* successfully analyzed this massive netlist and extracted all the gate-level STGs shown in Fig. 12 within 7 hours. It emphasizes that our proposed *FSMx-Ultra* framework is free from scalability issues while the existing FSM extraction techniques [26]–[29] suffer from such issues tremendously in analyzing such huge gate-level netlists. Additionally, we performed analysis on the hierarchical netlist of *OpenRISC 1200* as well. We observed that the same gate-level STGs, shown in Fig. 12, were successfully extracted by *FSMx-Ultra* in 5 h. 49 min. These two case studies demonstrate that *FSMx-Ultra* is a more promising solution than existing techniques in terms of run-time, performance, and scalability to extract control FSMs from the synthesized netlists (flattened or hierarchical) of industry-grade designs.

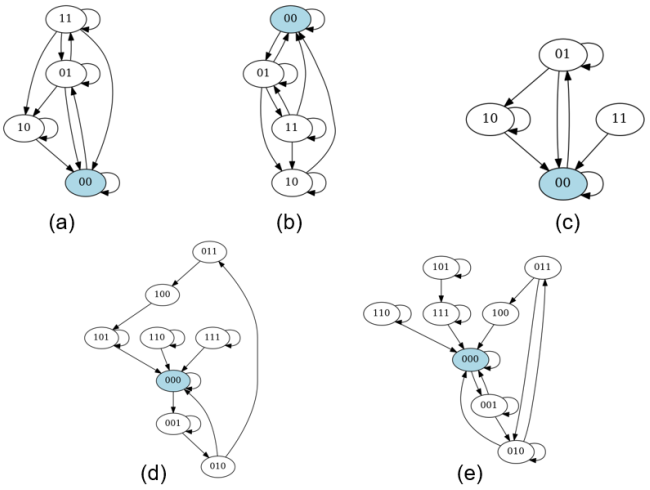


Fig. 12: The obtained 5 gate-level STGs of the control FSMs of *OpenRISC 1200* CPU [62]. All the control FSMs are encoded using the *Binary* state encoding. However, *FSMx* failed to extract the gate-level STGs since the associated state transition logic circuits contain a massive number of primary inputs.

VI. Applications of FSMx-Ultra

Our proposed *FSMx-Ultra* framework automatically detects all control FSMs present in a gate-level netlist with the corresponding gate-level STGs without any further manual analysis. The gate-level STGs of the control FSMs are generated in both textual and graphical representations and are human-readable. Therefore, these STGs can be utilized to reverse engineer the control flow of a complex SoC by an adversary. The attacker may quickly get an idea of the control FSMs' functionality in a design and model those FSMs at a higher abstraction layer. In conjunction, these gate-level STGs of the control FSMs can be used for the rapid verification of the control flow of an SoC after logic synthesis since *FSMx-Ultra* supports both hierarchical and flattened gate-level netlists and provide FSM extraction results much faster and more scalable compared to state-of-the-art schemes presented in [26]–[29].

The *FSMx-Ultra* framework can be highly efficacious for applications to ensure hardware security and trust effectively. First, the fault-injection assessment of the control FSMs present in a particular design in gate-level netlist abstraction has been proposed recently in [45]–[47]. Our proposed *FSMx-Ultra* framework can be easily extended to perform such a security assessment. In addition, *FSMx-Ultra* can also be used for performing information leakage assessment since the framework can identify the hidden states and transitions of the control FSMs which are absent in the RTL description of a design. Analyzing the extracted gate-level STGs, it can be easily verified whether the hidden don't-care states and transitions of the security-critical control FSMs assist in making an SoC design prone to information leakage issues via identifying the vulnerable state transitions of the FSMs.

Secondly, several FSM-based IP watermarking techniques have been proposed in existing literature [63]–[68]. Besides, numerous sequential logic locking schemes have been presented in [11], [28], [69], [70]. Precise recognition and extraction of all control FSMs and other relevant information present

in the synthesized gate-level netlist of an RTL design are crucial for such security applications as a major pre-processing phase, and *FSMx-Ultra* is a distinguishing candidate for this. The obtained control FSMs' gate-level STGs are handy for such an application since the *FSMx-Ultra* framework provides information on the state transition conditions, which can be utilized in developing watermarking and FSM-based logic locking schemes. This feature is quite similar to the *FSMx* framework [29]. However, *FSMx-Ultra* is better than the *FSMx* framework in terms of performance, scalability and general applicability as discussed in details in Section V.

Finally, apart from the applications for ensuring hardware security and trust, the *FSMx-Ultra* framework can be misused if it falls into the wrong hand. The proposed framework localizes the control FSM regions present in a highly unstructured gate-level netlist more precisely compared to existing approaches [26]–[28]. Thus, it may aid an adversary in launching powerful structural attacks on a synthesized gate-level netlist and performing malign activities. For instance, an attacker can implant malicious *Trojan* in a control FSM region of interest for bypassing particular state transitions and ultimately leaking sensitive information such as keys for cryptographic encryption and decryption operations [45]. The accuracy, run-time, and scalability of *FSMx-Ultra* can help tremendously in the localization phase of the control FSMs, thus will reduce the overall time required for performing a certain structural attack. Nonetheless, *FSMx-Ultra* can help the security engineers to evaluate the efficacy of a certain FSM-based logic locking technique from a defense perspective via analyzing the minimum time an attacker may take to localize all the control FSM regions in the unstructured gate-level netlist and hence launch powerful structural attacks. *FSMx-Ultra* is a more attractive solution compared to the proposed *FSMx* framework [29] for such an application.

VII. Conclusion

This paper proposes a fast, scalable, and precise technique based on state-of-the-art efficient graph algorithms and ATPG-based analysis to automatically recognize all the control FSMs present in the synthesized gate-level netlist of a particular RTL design with the corresponding human-readable gate-level state transition graphs. Experimental results on the synthesized gate-level netlists of several benchmark RTL designs varying in size and complexity have proved the efficacy of our proposed *FSMx-Ultra* framework in terms of performance, accuracy, and scalability, which is unfortunately absent in the state-of-the-art FSM extraction schemes. We intend to utilize *FSMx-Ultra* for performing fault-injection and information-leakage assessments in the post-synthesis gate-level netlist abstraction. Moreover, we envision incorporating *FSMx-Ultra* to develop novel sequential logic obfuscation and control FSM-based watermarking schemes in the future. To conclude, the *FSMx-Ultra* framework can be easily integrated into the concurrent VLSI design flow just after the logic synthesis stage. Existing FSM extraction techniques at the gate-level netlist abstraction suffer from scalability and accuracy issues. Therefore, *FSMx-Ultra* may open a new horizon in detecting security vulnerabilities present in a design, assisting rapid verification of the control flow of an SoC design after logic synthesis and aiding designers to take numerous security countermeasures for making an SoC design more secure at the pre-silicon stage of the state-of-the-art VLSI implementation flow.

References

[1] R. Karri et al., "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer, Vol. 43, No. 10*, pp. 39-46, IEEE, 2010.

[2] N. Farzana et al., "SoC security verification using property checking," *2019 IEEE International Test Conference (ITC)*, pp. 1-10, 2019.

[3] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *Annual International Cryptology Conference*, pp. 104-113, Springer, 1996.

[4] E. Biham et al., "Differential fault analysis of secret key cryptosystems," *Advances in Cryptology — CRYPTO '97*, pp. 513-525, Springer, 1997.

[5] P. C. Kocher et al., "Differential power analysis," *Annual International Cryptology Conference*, pp. 388-397, Springer, 1999.

[6] D. Hély et al., "Scan design and secure chip secure IC testing," *IOLTS: International On-Line Testing Symposium*, pp. 219-224, IEEE, 2004.

[7] M. Tehranipoor et al., "Integrated circuit authentication," *Switzerland: Springer, Vol. 10*, pp. 978-3, Springer, 2014.

[8] G. K Contreras et al., "Security vulnerability analysis of design-for-test exploits for asset protection in SoCs," *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 617-622, IEEE, 2017.

[9] S. Bhunia et al., "The Hardware Trojan War," *Switzerland: Springer*, Springer, 2018.

[10] D. Forte et al., "Hardware protection through obfuscation," Springer, 2017.

[11] K. Juretus et al., "Synthesis of hidden state transitions for sequential logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 40, No. 1*, pp. 11-23, 2020.

[12] M. T. Rahman et al., "CSST: Preventing distribution of unlicensed and rejected ICs by untrusted foundry and assembly," *2014 IEEE International symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT)*, pp. 46-51, IEEE, 2014.

[13] S. E Quadir et al., "A survey on chip to system reverse engineering," *ACM journal on emerging technologies in computing systems (JETC), Vol. 13, No. 1*, pp. 1-34, ACM, 2016.

[14] S. E Quadir et al., "State encoding watermarking for field authentication of sequential circuit intellectual property," *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 3013-3016, IEEE, 2012.

[15] M. Borowczak et al., "Mitigating information leakage during critical communication using S* FSM," *IET Computers & Digital Techniques, Vol. 13, No. 4*, pp. 292-301, IET, 2019.

[16] A. T Abdel-Hamid et al., "A survey on IP watermarking techniques," *Design Automation for Embedded Systems, Vol. 9, No. 3*, pp. 211-227, Springer, 2004.

[17] N. Anandakumar et al., "Rethinking Watermark: Providing Proof of IP Ownership in Modern SoCs,"*Cryptology ePrint Archive*, 2022.

[18] F. Farahmandi et al., "FSM Anomaly Detection Using Formal Analysis," *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 313-320, IEEE, 2017.

[19] J. Giomi, "Method of extracting implicit sequential behavior from hardware description languages," *US Patent 5,774,370*, 1998.

[20] M. E. Gilford et al., "Recognition of a state machine in high-level integrated circuit description language code," *US Patent 6,675,359*, 2004.

[21] https://opencores.org/, "OpenCores".

[22] https://github.com/freecores, "FreeCores".

[23] https://github.com/secworks, "SecWorks".

[24] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata Studies.(AM-34), Vol. 34*, Princeton University Press, 2016.

[25] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal, Vol. 34, No. 5*, pp. 1045-1079, 1955.

[26] K. S. McElvain, "Methods and apparatuses for automatic extraction of finite state machines," US Patent 6,182,268, Tech. Rep., 2001.

[27] Y. Shi et al., "A highly efficient method for extracting FSMs from flattened gate-level netlist," *Proceedings of 2010 IEEE international symposium on circuits and systems*, pp. 2610-2613, 2010.

[28] M. Fyrbiak et al., "On the difficulty of FSM-based hardware obfuscation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 293-330, 2018.

[29] R. Kibria et al., "FSMx: Finite State Machine Extraction from Flattened Netlist With Application to Security," *2022 IEEE 40th VLSI Test Symposium (VTS)*, pp. 1-7, 2022.

[30] http://www.ee.ncu.edu.tw/ jimmy/courses/DSD06/06_FSM.pdf.

[31] G. Vera et al., "Integrating Reconfigurable Logic in the First Digital Logic Course," 2006.

[32] T. Meade et al., "Gate-level netlist reverse engineering for hardware security: Control logic register identification," *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1334-1337, 2016.

[33] A. Nahiyan et al., "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2016.

[34] Z. Kohavi and N. K. Jha, "Switching and Finite Automata Theory," Cambridge University Press, 2009.

[35] https://www.allaboutcircuits.com/technical-articles/encoding-the-states-of-a-finite-state-machine-vhdl.

[36] T. Villa et al., "Synthesis of Finite State Machines : Logic Optimization," Springer, 1997.

[37] T. Villa et al., "Synthesis of Finite State Machines : Functional Optimization," Springer, 1997.

[38] C. Piguet, "Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools," CRC Press, 2018.

[39] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Scientific Computing, Vol. 1*, pp. 146-160, 1972.

[40] D. B. Johnson, "Finding All the Elementary Circuits of a Directed Graph," *SIAM Journal on Computing, Vol. 4, No. 1*, pp. 77-84, 1975.

[41] E. Nuutila et al., "On finding the strongly connected components in a directed graph," *Information processing letters, Vol. 49, No. 1*, pp. 9-14, Elsevier, 1994.

[42] B. Bilgin et al., "A more efficient AES threshold implementation," *International Conference on Cryptology in Africa*, pp. 267-284, 2014.

[43] T. D. Cnudde et al., "Higher-order threshold implementation of the AES S-box," *International conference on smart card research and advanced applications*, pp. 259-272, Springer, 2015.

[44] R. Ueno et al., "Toward more efficient DPA-resistant AES hardware architecture based on threshold implementation," *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pp. 50-64, Springer, 2017.

[45] A. Nahiyan et al., "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2016.

[46] A. Nahiyan et al., "Security-Aware FSM Design Flow for Identifying and Mitigating Vulnerabilities to Fault Attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 38, No. 6*, pp. 1003-1016, 2019.

[47] V. S. Rathor et al., "An energy-efficient trusted FSM design technique to thwart fault injection and trojan attacks," *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 73-78, IEEE, 2018.

[48] https://networkx.org/, "NetworkX".

[49] https://github.com/secworks/uart/tree/master/src/rtl, "UART Core".

[50] https://github.com/secworks/xtea/tree/master/src/rtl, "XTEA Cipher".

[51] https://opencores.org/projects/sayeh_processor, "SAYEH CPU".

[52] https://github.com/secworks/cmac/tree/master/src/rtl, "CMAC Cipher".

[53] https://github.com/secworks/sha256/tree/master/src/rtl, "SHA-256".

[54] https://github.com/secworks/sha512/tree/master/src/rtl, "SHA-512".

[55] https://github.com/secworks/poly1305/tree/master/src/rtl, "POLY-1305".

[56] https://github.com/secworks/aes/tree/master/src/rtl, "AES-128".

[57] https://github.com/gremerritt/multicycle-processor, "Tiny MIPS".

[58] https://github.com/wvangansbeke/Smart-Card-RSA, "Smart Card RSA".

[59] https://github.com/ultraembedded/core_usb_host, "USB HOST IP".

[60] https://github.com/freecores/mem_ctrl, "Memory Controller IP".

[61] https://github.com/YosysHQ/picorv32/blob/master/picorv32.v.

[62] https://github.com/openrisc/or1200, "OR1200 RISC Core".

[63] A. Cui et al., "A robust FSM watermarking scheme for IP protection of sequential circuit design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, No. 5*, pp. 678-690, 2011.

[64] M. Lin et al., "Watermarking technique for HDL-based IP module protection," *Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2007)*, pp. 393-396, 2007.

[65] A. T. Abdel-Hamid et al., "Finite state machine IP watermarking: A tutorial," *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)*, pp. 457-464, IEEE, 2006.

[66] K. Nguyen et al., "An FSM-based IP protection technique using added watermarked states," *2013 International Conference on Advanced Technologies for Communications (ATC 2013)*, pp. 718-723, IEEE, 2013.

[67] M. Lewandowski et al., "A novel method for watermarking sequential circuits," *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 21-24, 2012.

[68] R. Karmakar et al., "A cellular automata guided finite-state-machine watermarking strategy for IP protection of sequential circuits," *IEEE Transactions on Emerging Topics in Computing*, 2020.

[69] J. Kuai et al., "WaLo: Security Primitive Generator for RT-Level Logic Locking and Watermarking," *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 01-06, IEEE, 2020.

[70] A. Saha et al., "Oracall: An oracle-based attack on cellular automata guided logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 40, No. 12*, pp. 2445-2454, 2021.

Rasheed Kibria obtained his BS in Electrical Engineering from the Bangladesh University of Engineering and Technology (BUET) in 2019. Currently, he is a Ph.D. student in the Electrical and Computer Engineering department at the University of Florida, Gainesville, USA. His Ph.D. studies are funded by Defense Advanced Research Projects Agency (DARPA) and Dynetics. His research interest includes Hardware Security, Static Code Analysis, Gate-Level Netlist Analysis, Secure VLSI Design, and SoC Security Verification and Validation.

Farimah Farahmandi (S'13-M'18) is an Assistant Professor in the Department of Electrical and Computer Engineering (ECE) at the University of Florida (UF). She received her Ph.D. from the Department of Computer and Information Science and Engineering (CISE) at the University of Florida in 2018. She received her B.Sc. and M.Sc. from the Department of Computer Engineering at the University of Tehran, Tehran, Iran, in 2010 and 2013, respectively. Her research interests include design automation of System-on-Chips and energy-efficient systems, formal verification, hardware security validation, and post-silicon validation. Her research has been sponsored by SRC, DARPA, AFRL, DoD, Analog Devices, Ansys, and Cisco. Dr. Farahmandi is currently the associate director of Edaptive Computing Inc, Transition Center (ECI-TC) at the University of Florida.

Mark Tehranipoor (S'02-M'04-SM'07-F'18) is currently the Intel Charles E. Young Preeminence Endowed Chair Professor in Cybersecurity and ECE Department Chair at the University of Florida. His current research projects include hardware security and trust, supply chain security, IoT security, VLSI design, testing, and reliability. Dr. Tehranipoor has published over 500 journal articles and conference papers, delivered many talks, and published 13 books. He is a recipient of a dozen best paper awards and nominations, the 2008 IEEE Computer Society (CS) Meritorious Service Award, the 2012 IEEE CS Outstanding Contribution, the 2009 NSF CAREER Award, and the 2014 AFOSR MURI award. He received the 2020 University of Florida Innovation of the Year award. He serves on the program committee of more than a dozen leading conferences and workshops. He has also served as program chair of several IEEE and ACM-sponsored conferences and workshops (HOST, ITC, DFT, D3T, DBT, NATW, and more). He co-founded the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) and served as HOST-2008 and HOST-2009 General Chair. He is currently serving as a founding EIC for Journal on Hardware and Systems Security (HaSS) and Associate Editor for JETTA, JOLPE, IEEE TVLSI, and ACM TODAES. Before joining UF, Dr. Tehranipoor served as the founding director for CHASE and CSI centers at the University of Connecticut. He is currently serving as a founding director for the Florida Institute for Cybersecurity Research (FICS). Dr. Tehranipoor is a fellow of the IEEE, a golden core member of IEEE CS, and a member of ACM and ACM SIGDA.