Gate-Level Netlist Reverse Engineering Tool Set for Functionality Recovery and Malicious Logic Detection

Travis Meade*, Shaojie Zhang*, Zheng Zhao†, David Pan†, and Yier Jin‡
*Department of Computer Science, University of Central Florida
†Department of Electrical and Computer Engineering, University of Texas at Austin
‡Department of Electrical and Computer Engineering, University of Central Florida
travm12@knights.ucf.edu, zzhao.ee@gmail.com, dpan@ece.utexas.edu, shzhang@cs.ucf.edu, yier.jin@eecs.ucf.edu

***Abstract*—Reliance on third-party resources, including third-party IP cores and fabrication foundries, as well as wide usage of commercial-off-the-shelf (COTS) components has raised concerns that backdoors and/or hardware Trojans may be inserted into fabricated chips. Defending against hardware backdoors and/or Trojans has primarily focused on detection at various stages in the supply chain. Netlist reverse engineering tools have been investigated as an alternative to existing chip-level reverse engineering methods which can help recover functional netlists from fabricated chips, but fall short of detecting malicious logic or recovering high-level functionality. In this work, we develop a netlist reverse engineering tool-set which recovers high-level functionality from the netlist, thereby aiding malicious logic detection. The tool-set performs state register identification, control logic recovery and datapath tracking, which facilitates validation of encrypted/obfuscated hardware IP cores. Relying on 3-SAT algorithms and topology-based computational methods, we demonstrate that the developed tool-set can handle netlists of various complexities.**

I. Introduction

With an ever expanding integrated circuit production line, many parts of chip manufacturing occur overseas. As a result, assuring IC integrity and trustworthiness has become a challenge. Although there have been methods proposed to assist with chip-level verification, through testing and side-channel based hardware Trojan detection methods, existing solutions either rely on the availability of golden chips or lack detection accuracy [1]–[3]. Further, while most of these methods try to enhance the testing methods for security validation, there is a lack of reverse engineering tools which can rebuild the full functionality of the netlist for further analysis. Upon this request, various solutions and algorithms have been proposed trying to recover the data-path as well as the functionality of each circuit module in the data-path from a gate-level netlist. Existing solutions include behavioral pattern mining [4], word-level structure reconstruction [5], and structural and functional analysis on individual gates and sub-modules [6].

In this paper, we develop a novel gate-level netlist reverse engineering tool set for assisting users with the netlist verification process, including Reverse Engineering Logic Identification and Classification (RELIC), Reverse Engineering Datapath (REBUS) and Reverse Engineering Finite State Machine (REFSM). These tools can recover the functionality of the netlist for IC trust. Specifically, RELIC helps differentiate state registers from data registers [7]. REBUS can help reconstruct the datapath in a netlist. REFSM is an automated tool that utilizes a 3-SAT solver approach to explore and construct a FSM from a gate-level netlist [8]. The remaining of the paper is organized as follows: Detailed introduction to each tool within the tool-set will be provided in Section II to Section V. Demonstrations of the tool-set on selected netlists will be elaborated in the Section VI. Related work will be introduced in Section VII. Finally, conclusions are drawn in Section VIII.

II. Netlist Reverse Engineering Tool-Set

In this section, we will present a high level description of our methods which are the cornerstones of the developed netlist reverse engineering tool-set. A graphical user interface is also constructed so that users can easily use these tools for netlist analysis. Note that the tools we developed will not automatically identify and isolate the design flaws and/or malicious logic. Instead, the tool-set can assist users in gaining insight with respect to the functionality of an arbitrary netlist. This high level understanding can then help identify hardware Trojans, backdoors, and design flaws. The working procedure of these tools will be elaborated in the following sections.

A. RELIC

The RELIC tool is used for differentiating state registers from data registers. Given an arbitrary netlist, the RELIC tool will generate a score for each pair of wires that represents how similar their respective fan-in subgraph structures are. By using a novel graph structure matching algorithm, the RELIC tool generates possible wire pairs that have a high likelihood of either performing the same function within a replicated module or existing within the same data bus. That is, given any two signals in a netlist, the RELIC tool will attempt to match all fan-in wires of these two signals in such a way that the total similarity scores of these matchings is maximized. The whole process will run recursively until the maximized score is achieved or a preset depth has been reached to avoid infinite recursion. This matching is done using *min-cost-max* flow. The *memoization* technique is used in the RELIC algorithm to help reduce the runtime by preventing re-computation of known results.

B. REBUS

The second tool is the REBUS tool which is used to group individual signals into buses and generate the bus-level data flow within the netlist. The REBUS tool utilizes methods

similar to WordRev [5] to find potential word pairs. More specifically, the REBUS tool relies on structure similarity between signals to determine whether the given signals come from the same word. The key idea stems from the observation that internal signals within a bus have similar behaviors including similar gate functionalities, similar fan-in structures and similar controlling signals.

C. REFSM

The third tool is named REFSM which helps reconstruct the controlling logic of a netlist in the form of finite state machines (FSMs). The REFSM tool utilizes connectivity heuristics to find potential state registers and then employs 3-SAT solver strategies to construct the FSM from these identified state registers. For cases where submodules in a design have independent FSMs, REFSM will recover the global FSM from the flattened netlist. This FSM is the tensor product of two or more smaller FSMs. Therefore, a second step will be performed by the REFSM tool to decompose the global FSM into independent FSMs, representing the functionality of submodules.

III. RELIC

The main goal of reverse engineering netlist is classifying the role of each part of the netlist. The tool set begins with one broad classification of wires into two main groups: logic and data. We assume that wires can be further grouped into disjoint sets calledwords. These words are the abstracted words within the RTL (or similar higher level language used to represent circuit designs). Although detecting these words would be highly useful, the tool set settles for first finding logic wires.

A. Motivation

Wires from the same multi-bit word will likely undergo a similar function when propagating their data through the circuit (e.g. for statement `assign c = a + b;`, most wires in c will have identical functions). Because CAD tools are automated and follow a protocol, similar functionality synthesizes into similar wire-gate structures. This causes the wires that comprise words to have almost identical structure. Thus it is hypothesized that to determine if two wires are part of the same word, one can simply compare their structures.

Graph isomorphism has come a long way over the years, but for this problem it is desirable to have some error tolerance for comparing wire structures since some wire pairs within words might not be identically synthesized. This variation can occur when the function at bit level is not identical (e.g. no `carry` wire is used for the first bit of an adder) or the CAD tool utilizes varying gate structures to minimize area/power consumptions that are logically equivalent. The first is easy to detect using our proposed tool, but the later scenario is challenging.

In contrast wires associated with the netlist's logic tend to have a unique structure. Logic wires fall into two broad categories intended logic (e.g. registers and wires that are part of the specification) and unintended logic (e.g. hardware Trojans). The unintended logic can either be part of the generated logic (in the RTL prior to synthesis) or inserted logic (manually added after synthesis). The first type of unintended logic would structurally behave much like the intended logic, so it becomes more difficult to flag as a possible hardware Trojan.

Based on the findings mentioned above, we developed RELIC to assist users in identifying a netlist's logic controlling registers. RELIC aims to identify data words by finding wires that have a similar fan-in structure to other wires within the netlist.

B. Similarity Scores

Relying on the assumption that wires pair within words have a similar wire structure RELIC estimates structure similarity by generating a similarity score. Each score falls in the range of 0 to 1, where 0 represents no similarity and 1 represents high similarity. The method first recursively assess the similarity for each pair of fan-in wires between the original two wires in question. Following this the similarity scores are used to create a weighted bipartite graph. Each node of the created graph is a fan-in wire of the original two wires, and for each node pair from differing fan-in set is connected by an edge. The weight of each edge is the previously computed similarity score for the wires connected by it. After the graph is constructed weighted bipartite matching is used to generate the most probable set of related wire pairs. The returned score is the sum of the selected edges' weights divided by the maximum of the fan-in sizes. We compile the similarity scores and those wires with highly similar counterparts (e.g. wires with similarity scores above some predetermined value) are classified as datapath signals.

A pair of logic vertexes that each affect their respective outputs might not halt. This situation could cause the scoring procedure to infinitely call the scoring function on themselves. RELIC integrates a heuristic that causes the scoring function to break out early when a user defined depth is reached. When the break occurs we assume that the children of the two wires would match perfectly. Thus the resulting value becomes the size of the smaller fan-in divided by the size of the larger fan-in.

Memoization, a dynamic programming technique, is used to prevent similarity score re-computation. Wire pairs with large fan-out are particularly susceptible to duplicate queries, since for each wire pair in the fan-out of two wires a query will be performed. Memoization simply stores the previously computed value so wire pairs need only to be evaluated once.

IV. REBUS

Distinguishing data from logic is not the only goal of the tool set. A robust tool set should be capable of determining how data moves through the netlist. REBUS, developed with the goal of finding datapaths, utilizes forward propagation to assist in producing high level netlist expressions. REBUS' novelty comes from its wire pair word verification. The process of which is performed on the abstracted logical graph,

derived from the gate-level netlist. Moreover, the process is to generate the similarity score that was mentioned in Section III. In this situation the wires with similarity scores above a user defined threshold are consider similar.

V. REFSM

Similar to RELIC and REBUS, the REFSM tool retrieves high-level netlist description from gate-level netlists. However, REFSM focuses on control logic, by extracting the behavior of a netlist's logic affecting registers in the form of an FSM. Under REFSM's hood lies a variety of techniques: breadth first search (BFS) branch and bound, DPLL style 3-SAT solver, and a heuristic based tensor product decomposer. With a trained eye supervising the results, REFSM can be leveraged to detect malicious logic.

A. Performance Enhancement

Storing the full set of registers of a netlist for each state of the FSM can be slow and lead to very complicated FSMs that have very little meaning to the actual working mechanism of the logical FSM. The first step for improving both the results and performance is to determine what registers will be used to construct the FSM. Selection methods presented in this paper can be used, but barring the user from knowing the exact set of logic registers. Note that there will most likely be errors regardless of selection method. Luckily REFSM has some error correcting methods that will be described later.

The second step is to select the additional registers for shaping the FSMs transitions. In netlists, although some registers might not be part of the FSM state, they might have some affect on the next state in FSM. This leads to a register having an effect on the conditions and existence of state transitions. In a second situation, to check for corner cases of the netlist's logic, a user might have the desire to keep track of a non-FSM logic register without actually incorporating it into the FSM's state. Therefore, the support for hidden register is implemented in two ways. Firstly a user can specify a value, *k*. All registers that can affect the a state registers output value in fewer than *k* cycles is incorporated into the state without explicitly generating new output states. Secondly a user can keep track of a particular set of registers by providing the appropriate specification.

B. Breadth First Searching

To find all of the FSM's states and transitions a BFS over the FSM's states is used. The BFS is detailed in GETREGISTERSTATES in Algorithm 1. The first state in the BFS is the reset state, which is determined by applying the reset signal. All subsequent states are found using the 3-SAT solver. The proposition logical formulas are generated by applying the values of the current state of the BFS to the given gate-level netlist. Each register part of the BFS state have their own formula. Note that the states that are stored in the BFS are the ones that contain the non-FSM registers described earlier. The program stops once each found state is explored by the BFS.

Algorithm 1 Find all possible register states given a set of expressions $EXPS$ from a flattened netlist and a starting expression set $resetState$

```
1: function GetRegisterStates(EXPS, resetState)
2:     Add the resetState to the Queue; Set Seen to resetState
3:     while Queue ≠ ∅ do
4:         Get a currentState from Queue
5:         currentExp ← EXPS.LastState(currentState)
6:         F ← Fetch(currentExp)
7:         for nextState ∈ F do
8:             if nextState ∉ Seen then
9:                 Q.add(nextState)
10:                Seen ← Seen ∪ {nextState}
11:            end if
12:        end for
13:    end while
14:    return Seen
15: end function
16: function Fetch(exps)
17:    if exps contains no variables then
18:        return {exps}
19:    end if
20:    x ← first variable in exps
21:    newExps ← exps.set(x, false)
22:    F ← Fetch(newExps)
23:    newExps ← exps.set(x, true)
24:    F ← Fetch(newExps) ∪ F
25:    return F
26: end function
```

C. Boolean Satisfiability

For transition finding REFSM utilizes a DPLL style solver to determine potential next states. The details are in FETCH in Algorithm 1. The wire values that are not set by the BFS are the literals. This includes input and don't care registers. This method works well since many logical expressions for register are either large logical conjunctions (formed by large AND trees) or large logical disjunctions (formed by large OR trees).

D. Tensor Product Decomposition

REFSM has some extra capabilities for analyzing FSMs. FSM readability suffers when the method for determine the FSM registers fails to produce a single set of logic words. The FSM gets bogged down with many states and transitions, and Manual analysis of the recovered FSM becomes difficult. REFSM's decomposition can help correct errors made by the user or even speed up the process of netlist analysis.

The main goal of REFSM's decomposition is to separate potentially combined independent FSMs from each other. The resulting FSM is assumed to be a tensor product of the directed graphs (e.g. the FSMs). This first assumption is safe since in most cases combining FSMs acts as the tensor product of their state-transition graphs. Since Tensor decomposition is difficult when the graphs are directed, a heuristic is utilized. This is where REFSM decomposition makes its second assumption: each register affects the state of exactly one FSM. With the second assumption in mind the protocol initializes each register as belonging to its own "FSM". The algorithm checks each pair of "FSM"s for dependency. Two topological checks are utilized to verify the independence of the pair of FSMs.

The nodes (state) for the small FSM's tensor product is verified for the original FSM. In other words for each pair of individual states in the small FSMs' its combination state must exist somewhere in the original FSM. The other topology check is on the edges (transitions). It is similar to the node check in that for each pair of individual edges in the small FSMs' its combination edge must exist in the original FSM. If at least one node or edge does not exist, then the two register sets for the FSMs are replaced by their union. The process is repeated until all FSMs can be considered independent by the above check.

VI. Demonstrations

A. REFSM Benchmarks

To help show the potential of the tool-set, we ran the REFSM tool on netlists of various complexities. Using an Intel Core i5 dual-core processor clocked at 2.2 GHz with 16GB of RAM, the run-time of each netlist is shown in Table I.

TABLE I: Average Run-time for Sample Circuits

Testing Circuits	Registers	Total Gates	Time
RS232 Transceiver	59	168	120 ms
32-bit RSA	555	2139	1 s
MC8051 μP	578	6590	48 s

B. Control Logic Recovery on RS232 Transceiver

The RS232 transceiver includes two sub-modules for data transmitting and data receiving. Both the transmitter and the receiver work independently without interfering with each other. In addition, they have their own input/output pins at the top module. However, the flattened netlist does not maintain the circuit hierarchical structure and there is no clear boundary between them. Therefore, the selection of an RS232 circuit is ideal for verifying the capability of RE toolset in isolating different FSMs from a flattened netlist.

Using the flattened RS232 netlist as the input, our netlist reverse engineering toolset recovers the control logic in the format of FSM of the entire circuit. Figure 1 shows the recovered global FSM which contains 25 unique states with quite complicated transmission conditions among these states. This FSM, although containing the entire functionality of the RS232 circuit control logic, is almost meaningless to users and testers due to its complexity. However, the FSM decomposition algorithm can help simplify the FSM structure.

Using the recovered FSM in Figure 1, the developed FSM decomposition tool can isolate independent states from the entire FSM. In this case, two independent FSMs, Figure 2a and Figure 2b, are separated from the control logic in Figure 1. To validate the correctness of the FSM decomposition results, we build the real FSMs of the receiver and transmitter submodules in the RS232 circuit which are identical to the recovered FSMs both in available states and in all state transition conditions.

C. Graphical User Interface

To facilitate the netlist analysis process leveraging the developed tool-set, a graphical user interface (GUI) is built

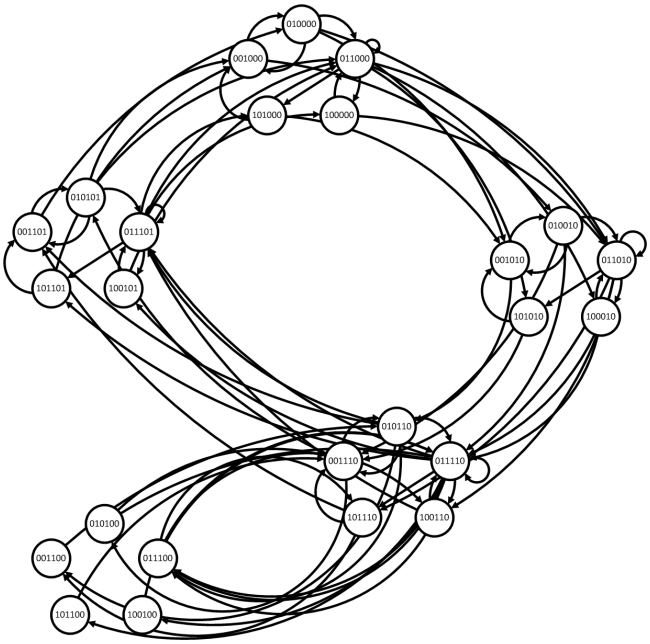


Fig. 1: Recovered Control Logic of the Entire RS232 Netlist

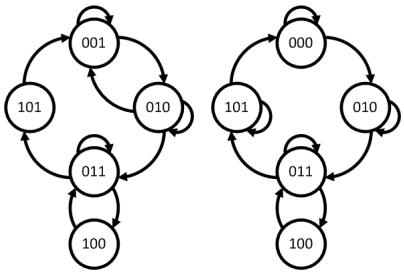


Fig. 2: The two FSMs recovered from the RS232 netlist. (a) First decomposed FSM and (b) second decomposed FSM.

(see Figure 3). All three tools, RELIC, REBUS and REFSM, are integrated into the GUI to provide a user-friendly working environment. Through the GUI, users can also monitor the intermediate results and adjust the parameters dynamically. For example, Figure 4a shows the steps to set up a reset signal within the RS232 Transceiver so that the netlist can be simulated starting from a reset state. Note that the reset selection can also be used to set values on all netlist signals. After a successful run of the REFSM tool the resulting FSM is given to the user, Figure 4b.

VII. Related Work

One of the central issues for reverse engineering digital circuits is determining state registers. State registers, or control logic, affect what a circuit will do with both the stored data on the chip and the data passed in through the inputs. Subtle manipulation of a circuit's state registers can be destructive and dangerous to users of these afflicted cores. Even worse, many chips today are produced by untrusted third parties to decrease the time-to-market (TTM). The major hurdle for recovering chips using brute force equivalence checking is the amount of time those methods take. In a situation where each register is

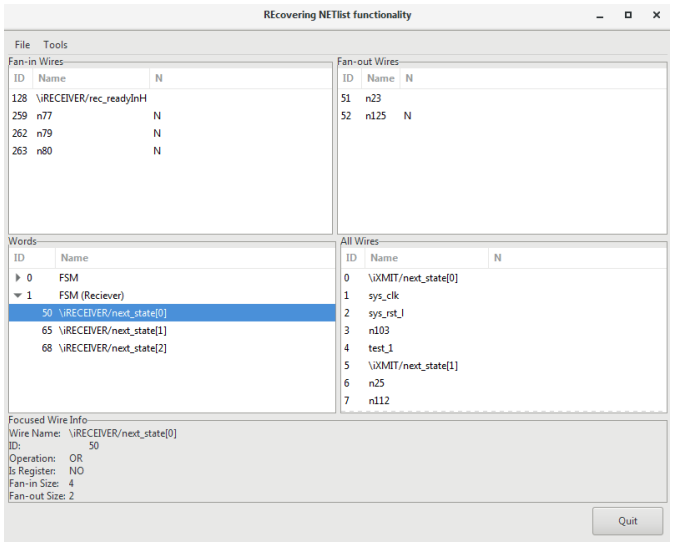
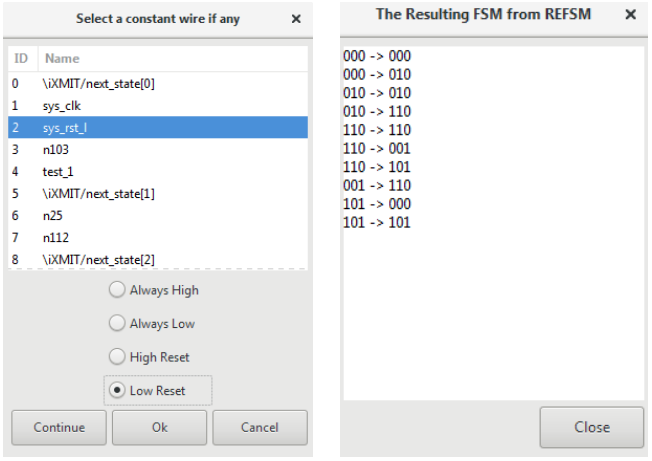


Fig. 3: Standard view of the graphical user interface.



(a) Reset wire dialog (b) Output dialog

Fig. 4: Dialog windows of the GUI.

independent and can either have '1' or '0' value, a chip with only 100 registers has over 10^{30} states.

Knowing the registers of the netlist's state logic is crucial to understanding the chip's functionality, malicious or not, many tools for specification recovery require having prior knowledge of the state registers [9]. To speed up these tools methods have been proposed for state register recovery. For example, the authors in [9], [10] utilized methods that classify a register as a state register if its output could indirectly affect itself. This technique reduces to classifying a register as logic if it is in a connected component. Although simple, it provided a good and fast baseline for determining logic registers. It should be noted, this technique fails to identify any data registers, if those registers are in the same connected component.

These methods show that there is a need for accurately identifying a netlist's important logic registers. To fulfill said need, this paper aims to classify registers quickly using a scoring function generated by examining the logical and topological similarity between pairs of registers in the netlist. Similar methods have been proposed that classify netlists, potentially infected with Trojans, based on a scoring metric [11]. This noteworthy paper also used structure checking to generate values for its metric. However, the method in [11] compared sub-structures of the given netlists to known Trojan structures. This reliance on a set of known structures could potentially fail to identify novel Trojans. Our tool, RELIC, removes the dependency of a structure library and classifies registers as logic or data based on a self-structure analysis in an attempt to detect potential novel Trojan structures, while maintaining an accuracy similar to other methods on known Trojan structures.

VIII. Conclusions

In this paper we demonstrated a netlist reverse engineering toolset including three separated but connected reverse engineering tools. A graphical user interface was also developed for easy implementation of the toolset. We also introduced the functionality of each tool within the toolset. For future work, we will investigate automating the whole netlist analysis process to further reduce the cost for high level functionality recovery.

References

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *IEEE Symposium on Security and Privacy*, 2007, pp. 296–310.

[2] M. Banga and M. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2010, pp. 56–59.

[3] A. Ferraiuolo, X. Zhang, and M. Tehranipoor, "Experimental analysis of a ring oscillator network for hardware trojan detection in a 90nm asic," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '12, 2012, pp. 37–42.

[4] W. Li, Z. Wasson, and S. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, 2012, pp. 83–88.

[5] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, 2013, pp. 67–74.

[6] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *Emerging Topics in Computing, IEEE Transactions on*, vol. 2, no. 1, pp. 63–80, 2014.

[7] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, "Gate-level netlist reverse engineering for trojan detection and hardware security," in *The IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1334–1337.

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

[9] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting fsms from flattened gate-level netlist," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 2610–2613.

[10] K. S. McElvain, "Methods and apparatuses for automatic extraction of finite state machines," U.S. Patent 6 182 268, 2001.

[11] M. Y. N. T. Masaru Oya, Youhua Shi, "A score-based classifiication method for identifying hardware-trojans at gate-level netlists," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ser. DATE '15, 2015, pp. 465–470.