Netlist Reverse Engineering for High-Level Functionality Reconstruction

Travis Meade[1], Shaojie Zhang[1], and Yier Jin[2]
[1]Department of Computer Science, University of Central Florida
[2]Department of Electrical and Computer Engineering, University of Central Florida
travm12@knights.ucf.edu, shzhang@cs.ucf.edu, yier.jin@eecs.ucf.edu

Abstract— In a modern IC design flow, from specification development to chip fabrication, various security threats are emergent. Of particular concern are modifications made to third-party IP cores and commercial off-the-shelf (COTS) chips where no golden models are available for comparisons. Toward this direction, we develop a tool, named Reverse Engineering Finite State Machine (REFSM), that helps end-users reconstruct a high-level description of the control logic from a flattened netlist. We demonstrate that REFSM effectively recovers circuit control logic from netlists with varying degrees of complexity. Experimental results also showed that the developed tool can easily identify malicious logic from a flattened (or even obfuscated) netlist. If combined with chip level reverse engineering techniques, the developed REFSM tool can help detect the insertion of hardware Trojans in fabricated circuits.

I. Introduction

Third-party resources in hardware circuit designs, mostly in the format of third-party fabrication services and third-party soft/hard IP cores for SoC development, are prevailingly used in modern circuit designs and fabrications. The availability of those resources largely alleviates the design workload, lowers the fabrication cost, and shortens the time-to-market (TTM). However, the heavy reliance on third-party resources/services also breeds security concerns. For example, a third-party IP core may contain malicious logic and/or design flaws which will be exploited by attackers after the IP cores are integrated into SoC platforms. In addition, a malicious foundry may insert hardware Trojans into the fabricated chips. The impact of malicious logic and design flaws in IP cores or fabricated chips threatens to ruin the credibility of vendors and places unnecessary security risks on the users.

To counter the threat of untrusted third-party resources/services, both pre-silicon and post-silicon trust evaluation approaches have been proposed. During the pre-silicon stage, researchers mostly focus on verification and validation methods on RT-level code. UCI [1] analyzes the RTL code to find lines of code that are never used in order to identify suspicious circuitry; however, hardware Trojans have been designed that successfully defeated UCI [2]. Other approaches for pre-silicon trust evaluation rely on formal methods, either to ensure the consistency between RTL code and high level specifications [3], or to ensure that the delivered IP cores fulfill pre-specified security properties [4, 5, 6, 7]. At the post-silicon stage, the majority of the trust evaluation and hardware Trojan detection methods rely on on-chip equivalence checking [8] or side-channel fingerprinting [9, 10]. Large design overhead and high testing cost is associated with these methods. 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, DARPA initiated the Integrity and Reliability of Integrated CircuitS (IRIS) program. The program emphasizes that the security challenges associated with third-party resources/services are coupled with the inability to guarantee the generation of comprehensive test vectors to test functions not in the specification [11]. In response to this program, 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, such as behavioral pattern mining [12], word-level structure reconstruction [13], and structural and functional analysis on individual gates and sub-modules [14].

While these proposed methods help recover the data-path and reconstruct the functionality of arbitrary gate-level netlists, the control logic, a less-regulated circuit component, is rarely discussed. The previously presented data-path functionality recovering methods cannot be used in control logic analysis for various reasons: 1) Signals are often in the format of multi-bit buses in a data-path, whereas they often act individually in control logic; 2) The full functionality of a data-path may be rebuilt through the analysis of separate gates and sub-modules. However, we have to recover the entire control logic, often in the form of finite state machines (FSMs), in order to understand the control logic functionality; 3) Due to the flexibility of FSM structures it is difficult to build a module library with all possible control circuit components. As a result, a control logic recovery method is required which, if combined with data-path analysis methods can help consumers to reconstruct the full functionality of the third-party IP cores (or fabricated chips) where RTL de-

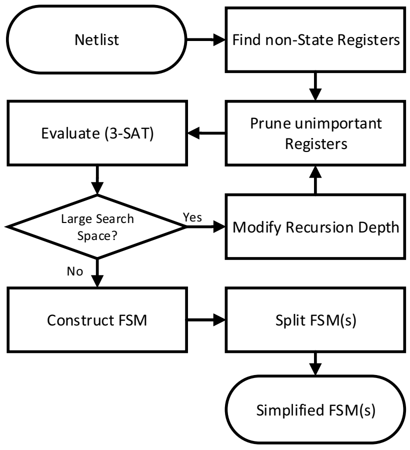


Fig. 1. REFSM Working Flow Diagram

scriptions are not available.

In this paper, an automated netlist analysis tool is developed to help users fully understand the circuit control logic without the need for consulting RTL descriptions. The automation tool is named Reverse Engineering Finite State Machine (REFSM) to emphasize its usage in rebuilding circuit control logic. As opposed to previous methods for FSM reverse engineering in [15, 16, 17], REFSM builds a Boolean expression based on the gate-level netlist related to the FSM registers and employs a 3-SAT solver to construct the FSM transition graph (3-SAT solvers have been used for FSM equivalence checking [18]). The main contributions of the paper include:

- The entire functionality of the control logic can be rebuilt on a high level description for any netlist so that any malicious behaviors can be identified easily;
- The use of heuristic algorithms helps solve the scalability issue and lead to large-scale circuit analysis;
- The analysis outcome will be automatically constructed to further reduce the testing time.

The rest of the paper is organized as following: Section II introduces the basic working flow and supporting algorithms of REFSM. Case studies of REFSM on various circuit designs are presented in Section III. Further experimentation results are elaborated in Section IV demonstrating the effectiveness and efficiency of REFSM in automatically detecting stealthy hardware Trojans. Concluding remarks are in Section V.

II. REFSM Working Flow

REFSM attempts to recover the control logic from a gate-level netlist and present to the user a higher-level description. A general outline of REFSM is shown in Figure 1. The netlist is first collected either from chip level reverse engineering or from the IP provider. The end user is then required to initiate the process and modify the recursion depth if run-time becomes an issue. Since designs can contain hundreds of thousands of gates or more, the first step is to reduce the number of gates to be analyzed by identifying and isolating FSM registers.

Algorithm 1 Find an FSM graph given a set of expressions $EXPS$ from a flattened netlist and a starting expression set $resetState$

```
1: function GetRegisterStates($EXPS, resetState$)
2:     Let $FSM$ be an empty graph $G(N, E)$
3:     Add the $resetState$ to the $Queue$; Set $N$ to $\{resetState\}$
4:     while $Queue \neq \emptyset$ do
5:         Get a $currentState$ from $Queue$
6:         $currentExp \leftarrow EXPS.LastState(currentState)$
7:         $F \leftarrow$ Fetch($currentExp$)
8:         for $nextState \in F$ do
9:             if $nextState \notin N$ then
10:                $Queue.add(nextState)$
11:                $N \leftarrow N \cup \{nextState\}$
12:            end if
13:            $E \leftarrow E \cup \{(currentState, nextState)\}$
14:        end for
15:    end while
16: return $FSM$
17: end function
18: function Fetch($exps$)
19:     if $exps$ contains no variables then return $\{exps\}$
20:     end if
21:     $x \leftarrow$ first variable in $exps$
22:     $newExps \leftarrow exps.set(x, false)$
23:     $F \leftarrow Fetch(newExps)$
24:     $newExps \leftarrow exps.set(x, true)$
25:     $F \leftarrow Fetch(newExps) \cup F$
26: return $F$
27: end function
```

A. Create Logic Graph and Identify State Registers

REFSM starts by creating the logical graph from a flattened netlist. The graph contains edges from inputs/registers to registers/outputs. Since REFSM determines the potential states of the registers, the outputs will not be considered. Any logic that is output exclusive is removed from the graph. What remains is logic from inputs and registers that can affect other registers either directly (register at time t can vary from register state/input at time $t-1$) or indirectly (register at time t may vary based on register state/input at time $t-k$, where $k>1$). REFSM then identifies potential state registers following the heuristic algorithms proposed in [17].

B. Prune Graph

Next, using the netlist and the set of state registers, REFSM prunes out potentially unimportant registers. The process involves a Breadth First Search (BFS) through the netlist up to a maximum distance of δ from the set of state registers[1]. This precomputation is used to produce a smaller subset of the netlist, which allows for an estimated register state graph in a reasonable amount of time and memory usage. However, in case that the current δ still causes program problems, δ will be decreased by user to run the algorithm again. The δ reduction process is performed until a state register graph is produced.

The justification for graph pruning is as follows. Some data registers are required for determining which states

[1]Similar to the step of deciding which registers are state registers, the register pruning is a heuristic approach.

are visited. Even though they might not affect the state registers immediately, they can cause significant changes to state register values in the future. Conversely, some of the registers might not be pertinent to what state the circuit is in or can visit. As an example, a register only affects outputs and, unless considered a state register, can be removed since it does not affect state registers. The call to remove registers is tough, so all registers are considered important. Only if the amount of possible states becomes too large, REFSM will prune some potentially less important registers. Our implementation considers both '0' and '1' as potential values for each "unimportant" register. If there are 10 registers that are not considered important by the pruning step, then each state actually corresponds to over 1000 states. Checking and storing each one of these can take time, but certain assumptions about the graph can also reduce the number of states that need to be considered. The process of checking and pruning is performed until the number of states is small enough that the state graph can be fully constructed. Analysis can then be performed on the resulting graph to recover control flow and/or to detect malicious logic.

C. Evaluate State Space

After generating a pruned graph, REFSM searches for all possible states of registers that are achievable by using the function GetRegisterStates (see Algorithm 1). The given netlist is represented by a set of Boolean logical expressions, $EXPS$, and a set of false and true values ('0' and '1') to represent each state that the registers can take on. The only registers that are listed in each state are those which were determined to be important in the prune step. The *Queue* is initialized with the reset state (*resetState*). Meanwhile, the set of seen states (N) also contains the reset state to prevent reusing it again. By looping through all elements in the *Queue* all possible register states are generated. A single iteration starts by pulling out the first element in the *Queue*. A new set of expressions is generated by filling in all the values currently in the register state. As an example, if the register is set to be true (value '1') in the current state, then when making the new expressions from the netlist all variables relying on the register's output will be recalculated accordingly. This new expression is sent into the 3-SAT function, Fetch, for evaluation and returns the set of all achievable register states using the given expression. The GetRegisterStates function constructs an FSM graph by searching for any states not included in the graph, and then evaluating which states they can reach. Each new state is added both into the *Queue* and into N. The overall run-time is O($|N|^2 + |N| \times 2^{(\#inputs)}$).

As a key part of the function GetRegisterStates, the Fetch function starts by checking the expression for unassigned variables. If there is a variable that has yet to be assigned and the variable can affect the outcome of the expression, the Fetch function will need to decide what value to use. Otherwise, it will return the expression as it is. If there were unassigned variables, the Fetch function will randomly pick one of them, set its value to '0', check the outcome recursively and add it into the resulting expressions. The Fetch function will then set the variable to '1', check the outcome, and add the resulting expression into the output. After going through all variables, the function will then return all identified states.

The complexity of the Fetch function operation is $O(2^n)$ in the worst case, where n is the number of variables that can change. In practice, due to the structure that many netlists follow, there are few variables that have an effect on the outcome of the next state. Most of the states terminate at a depth of 8 or less in our experimentation (See Section III). This makes the number of visited states less than 256. Further, many of the inputs perform a similar function so if one is set to '1', the others no longer need to be checked. For example given 20 variables ANDed or ORed together, the number of decisions that need to be made becomes 21. Although the computational complexity of the Fetch function appears daunting, it normally can be run in a reasonable amount of time such that the total run-time for REFSM becomes very low (See Table I).

D. Post-Processing on Reconstructed FSM

After deriving the global FSM, some extra steps for further analysis of the recovered control logic may be required. Determining simple transition conditions is one task that REFSM performs. This enables users to find suspicious transitions. A more important task is separating local FSMs from the global FSM, which is referred as FSM decomposition and is described below.

For demonstration purpose, we consider the case that two independent FSMs were merged. This results a pair of states (α_i, β_j) of the merged FSM, where α_i is from the first FSM and β_j is from the second FSM. Each pair of transitions that originate from the individual states should be traversable. The edges leaving the state (α_i, β_j) will contain at least the Cartesian product of the reachable states from state α_i and β_j. More formally

$$\{(\alpha_{i'}, \beta_{j'}) \mid \alpha_{i'} \in E_1[\alpha_i] \wedge \beta_{j'} \in E_2[\beta_j]\} \subseteq E_{(1\times 2)}[(\alpha_i, \beta_j)] \quad (1)$$

where $E_F[\alpha]$ is the state set that can be reached from state α in an FSM F. This infers that the merged FSM will be the tensor product of the original FSMs.

It should be noted that there have been algorithms which can decompose the tensor products on undirected, unlabeled, connected graphs into unique prime factor decompositions (UPFD) in polynomial time [19]. However, to decompose a merged FSM involves directed graphs and appears to be a harder problem. Therefore a heuristic-based approach is used to take advantage of the register labeling to split the graph into UPFD. The bottom part of Figure 1 presents an overview of the decomposition heuristic used in REFSM. The basic idea is to assume that each pair of registers is originally independent. Then look

TABLE I
Average Run-time for Sample Circuits

Testing Circuits	Registers	Total Gates	Time
RS232 Transceiver	59	168	1 s
32-bit RSA	555	2139	< 1 s
MC8051 μP	578	6590	39 s
SPARC μP	119911	232978	600 s

for contradicting sets of independent registers (either by vertex label or transition topology) and merge the found sets together until all register sets can properly construct the original FSM using their tensor product. Algorithm 2 lists the detailed description of the used algorithm.

Algorithm 2 Returns a partition of an FSM given a set of registers, R, and an FSM graph $G(N, E)$

```
 1: function SplitFSM($R$, $G(N, E)$)
 2:     Let $\mathbb{P} = \{P_i | P_i$ is the Partition containing register $i\}$
 3:     Assume no register depends on a register other than itself.
 4:     for $i, j \in R$ such that $P_i \neq P_j$ do
 5:         Let $G_i(N_i, E_i)$ be the FSM dependent on $i$
 6:         Let $G_j(N_j, E_j)$ be the FSM dependent on $j$
 7:         Let $G'(N', E')$ be the FSM dependent on $i$ and $j$
 8:         if there exists $u \in N_i$ and $v \in N_j$ and $(u, v) \notin N'$ then
 9:             $P_i \leftarrow P_i \cup P_j$; $P_j \leftarrow P_i$
10:         else
11:             if there exists $e \in E_i$ and $l \in E_j$ and $(e, l) \notin E'$ then
12:                 $P_i \leftarrow P_i \cup P_j$; $P_j \leftarrow P_i$
13:             end if
14:         end if
15:     end forreturn $\mathbb{P}$
16: end function
```

III. Experimental Results

In order to verify the effectiveness and the scalability of the developed REFSM tool, we applied the tool on various circuit designs ranging from small-scale ASIC designs to medium- and large-scale microprocessors. As we will demonstrate shortly, the control logic within all these testing circuits are recovered successfully in the format of finite state machines. The experimental tests are run on a desktop with Intel i7 quad-core and 16GB memory. The average run-time for different circuits are listed in Table I. For small-scale and medium-scale circuits, our algorithm can reconstruct the circuit control logic from a flattened netlist in less than 1 minute (most the time less than 1 second). The run-time is below 10 minutes even for large-scale circuits. From Table I, we can also find that in general the REFSM would have a larger computation time for larger circuits. However, the complexity of the control logic will affect the computation time. For example, 32-bit RSA Encryption[20] circuit finishes faster than the smaller RS232 transceiver due to the RSA circuit's more regular circuit structure.

A. RS232 Transceiver

The RS232 transceiver includes two sub-modules for data transmitting and data receiving. The sub-modules including 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 REFSM in isolating different FSMs from a flattened netlist.

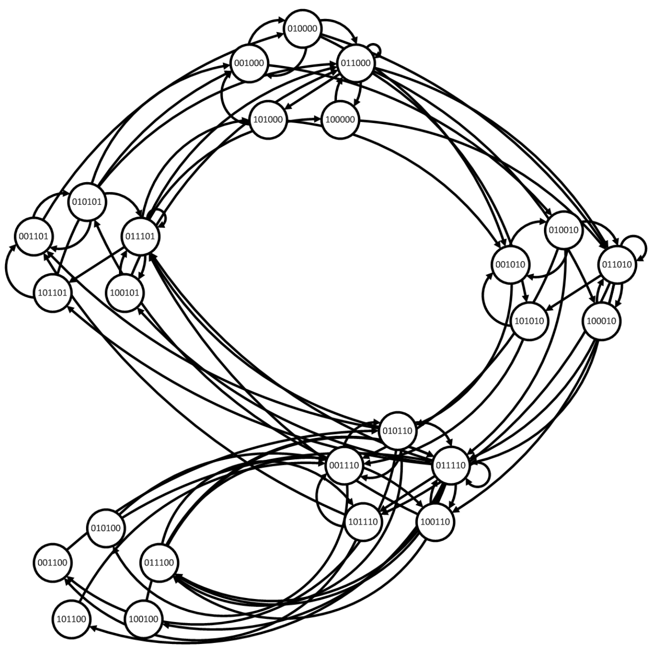


Fig. 2. Recovered Control Logic of the Entire RS232 Netlist

Using the flattened RS232 netlist as the input, our REFSM tools recover the control logic in the format of FSM of the entire circuit. Figure 2 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 component of REFSM can help simplify the FSM structure.

Using the recovered FSM in Figure 2, the developed FSM decomposition tool can isolate independent states from the entire FSM. In this case, two independent FSMs, Figure 3a and Figure 3b, are separated from the control logic in Figure 2. 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.

B. 8051 Microprocessor

The reason we used the 8051 microprocessor is to show the potential of REFSM in dealing with a highly-complex circuit structure. The source code of the 8051 microprocessor is written in VHDL, where each instruction will take up to three clock cycles to complete [21]. Based on

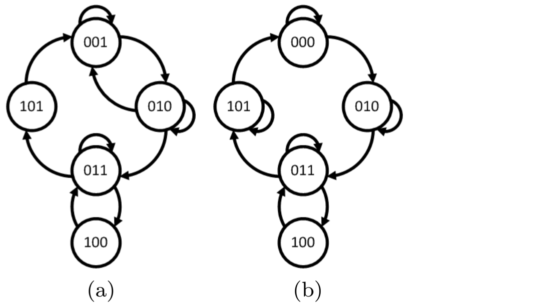
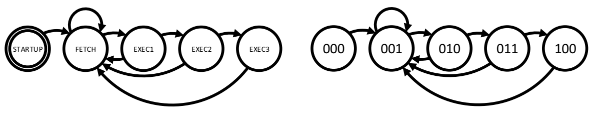


Fig. 3. The two FSMs recovered from the RS232 netlist. (a) First decomposed FSM and (b) second decomposed FSM.



(a) The RTL FSM. (b) The REFSM FSM.

Fig. 4. The FSM Recovered from MC8051 Netlist and RTL.

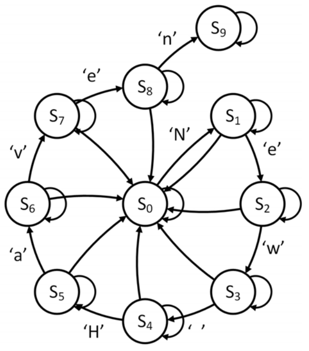


Fig. 5. The Recovered Hardware Trojan Trigger

the RTL code, we first constructed the real FSM when dealing with different instructions (see Figure 4a). We then synthesize the circuit and generate the flattened netlist of the 8051 microprocessor. The flattened netlist is then used as the input of the REFSM, which then recovers the control logic from the netlist. The recovered netlist is shown in Figure 4b. A comparison between Figure 4a and Figure 4b shows us that these two FSMs are of the same structure. In fact, the transition conditions are also identical.

IV. REFSM in Hardware Trojan Detection

The capability of REFSM for control logic recovery can also help detect hardware Trojans which are triggered by a specific input sequence, so-called sequential Trojans. Compared to the hardware Trojans that rely on only combinational logic to be triggered, sequential Trojans are much more difficult to activate and can evade many hardware Trojan detection methods [22]. However, since the behavior of the sequential Trojan triggering mechanism can be modeled as an FSM with the specific input sequence serving as the transition conditions, REFSM can help rebuild and isolate the Trojan FSM. From this circuit users/testers can easily identify the Trojan logic as well as the Trojan triggering conditions. For demonstration purposes, a Trojan-infected cryptographic platform is used [23, 24]. The platform is an FPGA implementation designed to perform all necessary operations for cyphertext transmissions through public channels. The user inputs data via a keyboard attached to a PS2 interface. This text is displayed through a VGA port onto an attached monitor. The user then initiates the encryption of the data entered via a button on the FPGA board. The encryption used is an 128-bit AES encryption core; the user also has the ability to select up to 16 different encryption keys by changing a combination of four switches on the FPGA before initiating the encryption sequence. Once encryption is finished, the user can then send the encrypted data through an on-board serial port.

In this design a Trojan was inserted in the top level module that uses a finite state machine to read a specific input sequence from the user, via the keyboard. Once the sequence is entered, the activated hardware Trojan will leak the AES encryption key through the serial port. The Trojan trigger seems simple but it can evade many hardware Trojan detection methods [24].

However, if we can identify all states of the Trojan FSM, determining the the actual behavior of the Trojan becomes apparent. Using the state space exploration techniques presented, all FSM states and transitions were correctly identified by the REFSM, as well as the correct conditions of the inputs for each transition. State diagrams were constructed of the edge-lists for the recovered FSMs. Figure 5 shows the recovered FSM of the inserted hardware Trojan and its triggering conditions. The letter on each transition curve shows the keyboard input which will enable the transition among these states. While the REFSM tool will not tell us whether the recovered FSM is genuine or malicious, users/testers can easily identify the suspicious logic and conclude that the special input sequence, 'New Haven' in this case, is outside the design specification and therefore potentially a hardware Trojan trigger. Users may validate their findings by triggering the suspicious circuit by inputting the special sequence.

Besides the elaborated example, we also applied our solutions to the hardware Trojan benchmarks from Trust-Hub [20]. Table II shows some of the testing results from which we can find that the REFSM tool can help detect hardware Trojans with sequential trigger and/or sequential payload in seconds.

V. Conclusion

This paper proposed and evaluated a method for reverse engineering the control logic from a gate-level netlist. The algorithm designed and implemented showed promising results with reasonable run time on standard desktop computer hardware. For every test, all states were successfully identified along with their correct state transitions and conditions leading to near perfect FSM recon-

TABLE II
Run-Time and Trojan Detection Capability on Trust-Hub Benchmark

Benchmark	Trigger	Payload	Trojan Detected?	Run-time
AES-T100	Always On	CDMA Trojan Side Channel	Detected	18 s
AES-T400	Plaintext = 128'hffffffffffffffffffffffffffffffff	RF Trojan Side Channel	Detected	< 1 s
AES-T800	Plaintext = 1) 128'h3243f6a8885a308d313198a2e0370734 2) 128'h00112233445566778899aabbccddeeff 3) 128'h0 4) 128'h1	CDMA Trojan Side Channel	Detected	< 1 s
b15-T400	Address = 8'hFF	Denial of Service	Detected	< 1 s
s38584-T100	Scan Enable Mode	Design Malfunction	Detected	< 1 s
MC8051-T200	pcon (control_mem) = 1'b1	Reduced Design Reliability	Detected	90 s

struction. In addition, the developed tool helps identify sequential hardware Trojans which, otherwise, would be very difficult to detect through existing testing methods. We expect that the developed tool will be widely implemented in other hardware security areas. Nevertheless, one shortcoming of the developed tool stems from the fact that all tests were to compare the FSM implemented in RTL source code with the recovered FSM to verify the correctness. Manually analyzing the RTL source code to construct a FSM can lead to possible errors or incomplete state spaces. Therefore, more work will be done to automate the verification process for the REFSM tools.

References

[1] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proceedings of IEEE Symposium on Security and Privacy*, 2010, pp. 159–172.

[2] C. Sturton, M. Hicks, D. Wagner, and S. King, "Defeating UCI: Building stealthy and malicious hardware," in *2011 IEEE Symposium on Security and Privacy (SP)*, 2011, pp. 64–77.

[3] 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.

[4] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[5] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *IEEE 30th VLSI Test Symposium (VTS)*, 2012, pp. 252–257.

[6] ——, "A proof-carrying based framework for trusted microprocessor IP," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.

[7] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 99–106.

[8] D. Lin, S. Eswaran, S. Kumar, E. Rentschler, and S. Mitra, "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15, 2015, pp. 1168–1173.

[9] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.

[10] C. Lamech, R. Rad, M. Tehranipoor, and J. Plusquellic, "An experimental analysis of power and delay signal-to-noise requirements for detecting Trojans and methods for achieving the required detection sensitivities," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1170–1179, 2011.

[11] DARPA, Microsystems Technology Office, "Integrity and reliability of integrated circuits (IRIS)," 2010.

[12] 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.

[13] 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.

[14] 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.

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

[16] L. Yuan, G. Qu, T. Villa, and A. Sangiovanni-Vincentelli, "An fsm reengineering approach to sequential circuit synthesis by state splitting," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 6, pp. 1159–1164, 2008.

[17] 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.

[18] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using sat for combinational equivalence checking," *Design, Automation and Test in Europe, 2001.*, pp. 114–121, 2001.

[19] W. Imrich, "Factor cardinal product graphs in polynomial time," *Discrete Mathematics*, vol. 192, no. 1-3, pp. 119–144, 1997.

[20] *https://www.trust-hub.org/.*

[21] Oregano Systems, "8051 IP core," http://www.oreganosystems.at/?page_id=96.

[22] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, and Y. Jin, "FIGHT-metric: Functional identification of gate-level hardware trustworthiness," in *Design Automation Conference (DAC)*, 2014.

[23] *http://isis.poly.edu/esc/2008/index.html.*

[24] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57.