INTEGRATION the VLSI journal 58 (2017) 563–570

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

IP protection through gate-level netlist security enhancement

Travis Meade[a], Shaojie Zhang[a], Yier Jin[b,*]

[a] Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA
[b] Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA

ARTICLE INFO

Keywords:
Hardware security
Gate-level netlist
IP protection

ABSTRACT

In modern Integrated Circuits (IC) design flow, from specification to chip fabrication, various security threats are emergent. These range from malicious modifications in the design, to the Electronic Design Automation (EDA) tools, during layout or fabrication, or to the packaging. Of particular concern are modifications made to third-party IP cores and commercial off-the-shelf (COTS) chips where no Register Transfer Level (RTL) code or golden models are available. While chip level reverse engineering techniques can help rebuild circuit gate-level netlist from fabricated chips, there still lacks a netlist reverse engineering tool which can recover the full functionality of the rebuilt netlist. 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 show that the REFSM can easily identify malicious logic from a flattened (or even obfuscated) netlist. Supported by REFSM, another tool, called Reverse Engineering Hardware Obfuscation for Protection (REHOP), is developed to enhance gate-level netlist security without learning the RTL code.

1. Introduction

Third-party resources in hardware circuit designs, mostly in the format of third-party fabrication services and third-party soft/hard Intellectual Property (IP) cores for System on Chip (SoC) development, are prevalently used in modern circuit designs and fabrications. The availability of third-party soft/hard IP core 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 third-party 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 [4–7]. 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 [12,13], word-level structure reconstruction [13,14], and structural and functional analysis on individual gates and sub-modules [15]. Many of these methods although formal lack the ability to extract control logic. Worse even, these methods tend to be

* Corresponding author.
E-mail addresses: travm12@knights.ucf.edu (T. Meade), shzhang@cs.ucf.edu (S. Zhang), yier.jin@eecs.ucf.edu (Y. Jin).

http://dx.doi.org/10.1016/j.vlsi.2016.10.014
Received 11 May 2016; Received in revised form 4 August 2016; Accepted 12 October 2016
Available online 05 November 2016

limited by the fact that they require prior knowledge of the design sub-modules to be effective, and thus although these methods are formally defined they can still suffer from inaccuracies.

While these proposed methods help recover the functional blocks in the data-path and reconstruct part of 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 descriptions are not available.

Orthogonal to the above mentioned IP validation methods, researchers are also investigating defensive solutions among which netlist obfuscation for IP protection serves as a leading example [16]. Due to the uncertainty of de-obfuscation tools and assumptions about the behavior of adversaries these methods appear to be effective. These methods insert a small FSM, which requires traversal before access to the real functional FSM is granted. Adversaries attempting to directly rip off the IP without concern to the unlocking sequence encounter issues. In practice, the infrequency with which the edges are traversed ($2^{|I|}$ per edge, where I is the set of input wires that are used to unlock the functional mode of the FSM) prevents an adversary from accessing the real FSM.

However, with the use of something as simple as a scan chain the adversary gains a tremendous amount of Reverse Engineering capability. An adversary can determine the required input for an edge traversal in roughly 1000 vectors, but can then remember this required key, which reduces the expected number of test vectors from 10^{12} to 4000. The protection provided by these methods deteriorates even faster when the adversary knows the gate-level netlist, even if the designed is flattened [16].

Upon these challenges in IP protection, the goal of this paper is of two-fold. First, an automated netlist analysis tool is developed to help users fully understand the circuit control logic without the need for consulting the RTL description. 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 [17–19], 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 registers transition graph (3-SAT solvers have been used to check equivalence of FSMs [20]). Second, leveraging the REFSM tool, a new gate-level netlist security enhancement framework is developed to protect a third-party netlist without the assistance of an RTL description, nor its high level specification. The new framework, called Reverse Engineering Hardware Obfuscation for Protection (REHOP), is demonstrated herein to outperform existing netlist obfuscation methods [21,22]. The main contributions of this paper include:

- We present REFSM, a reverse engineering tool that can fully recover the control logic from a gate-level netlist. REFSM produces a readable, high-level description of the gate-level netlist making maliciously inserted logic easily identifiable.
- We produce and demonstrate an automated REFSM toolchain.
- We present REHOP, a gate-level netlist control logic obfuscation tool. We demonstrate it is resilient to existing de-obfuscation techniques.
- We produce and demonstrate an automated REHOP toolchain integrated with REFSM.

The rest of the paper is organized as follows: Section 2 introduces the basic working flow and supporting algorithms of REFSM. Case studies of REFSM on various circuit designs are presented in Section 3. Further experimentation results are elaborated in Section 4 demonstrating the effectiveness and efficiency of REFSM in automatically detecting stealthy hardware Trojans. Section 5 introduces the REHOP tool for netlist security enhancement and its effectiveness in IP protection. Concluding remarks are in Section 6.

2. 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 Fig. 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. Once identified, REFSM explores the complete state space of the FSM registers using a 3-SAT solver and determines all FSM register states that can be achieved from a starting state. Performing the 3-SAT solver multiple times enables the user to construct a FSM state register transition graph from which the general topology of the finite state machine is gathered. To do this, the end user may need to adjust the recursion depth as desired, or keep on running the solver until the desired results are obtained. The selection of the recursion depth may vary depending on the specification of the circuit-under-test. Using the reset state of the FSM as a starting point, the FSM graph is then explored to provide an order of the state transitions so that generating a higher level description of the FSM can be done more easily. To elaborate the details of the REFSM working flow in Fig. 1, we will first address the key ideas behind each stage of the REFSM working flow. Technical details which are essential to accurate FSM control logic recovery will be introduced later.

REFSM was constructed based on the following three key ideas.

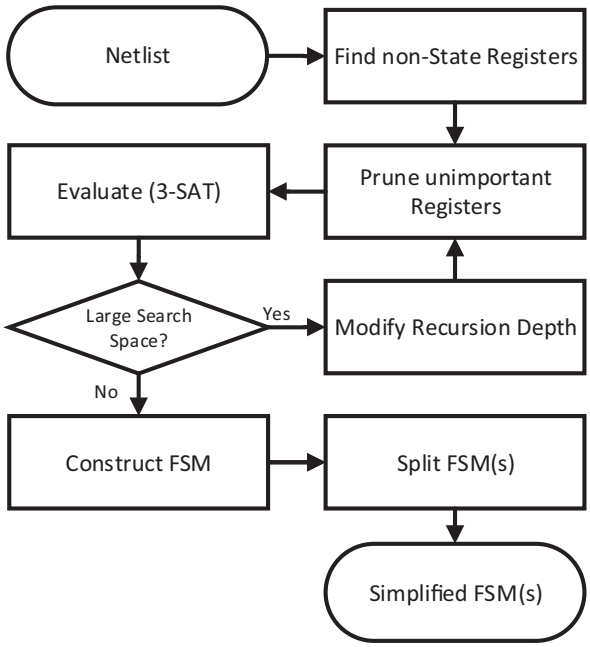


Fig. 1. REFSM working flow diagram.

First, since storing all possible circuit states and transitions among all states will cause a massive memory overhead, as a preliminary step, a subset of registers is selected to represent the achievable states. Selecting the subset of registers has proved to be difficult. We will leverage existing methods to identify state registers. Meanwhile, in order to gain a better understanding of the behavior of the whole netlist, an expanded set of registers may be needed. This expanded set will then be determined based on the desired accuracy as well as consideration of the runtime and memory usage.

After we get the state registers, the second step is to derive all transitions between each circuit state. In this case, since a netlist can be directly represented by 3-SAT expressions, solving the transition graph of the netlist can be converted into a 3-SAT problem. Therefore, an efficient 3-SAT algorithm will be sufficient for extracting the transition graph from a netlist.

Third, the recovered global FSM may not be the one users are looking for since the combination of different FSMs from submodules will make it difficult to explain the overall functionality. That is, a target netlist may have multiple independent modules combined together with each module containing their own control logic in the form of FSMs. As the result, the third step is to decompose the global FSM into sub-FSMs for each submodule. To achieve this goal, a straightforward heuristic algorithm is applied which can help decompose the global FSM with high computing efficiency and reduced runtime.

2.1. 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 (registers whose fan-out contains no registers) 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 the potential state registers following the heuristic algorithms proposed in [19].

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 ≠ ∅ do
5:      Get a currentState from Queue
6:      currentExp ← EXPS. LastState(currentState)
7:      F← FETCH (currentExp)
8:      for nextState ∈ F do
9:      if nextState ∉ N then
10:     Queue. add (nextState)
11:     N ← N ∪ {nextState}
12:     end if
13:     E ← E ∪ {(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
20:     return {exps}
21:     end if
22:     x← first variable in exps
23:     newExps ← exps. set (x, false)
24:     F ← Fetch(newExps)
25:     newExps ← exps. set (x, true)
26:     F ← Fetch(newExps) ∪ F
27:     return F
28: end function
```

2.2. 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 the program problems, δ will be decreased by users and the algorithm will be run 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 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.

2.3. 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 runtime is $(O|N|^2 + |N| \times 2^{(\#inputs)})$.

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

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 function output. After we go through all of the 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 3). 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 1).

2.4. 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. In this case, 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

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

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 [23]. 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 Fig. 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 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 (*! ht*). 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 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 ∈ R such that P_i ≠ P_j do
5:    Let G_i(N_i, E_i) be the FSM with registers dependent on i
6:    Let G_j(N_j, E_j) be the FSM with registers dependent on j
7:    Let G'(N', E') be the FSM with registers dependent on i and j
8:    If there exists u ∈ N_i and v ∈ N_j and (u, v) ∉ N' then
9:    P_i ← P_i ∪ P_j; P_j ← P_i
10:   else
11:   If there exists e ∈ E_i and l ∈ E_j and (e, l) ∉ E' then
12:   P_i ← P_i ∪ P_j; P_j ← P_i
13:   end if
14:   end if
15:   end for
16:   return ℙ
17: end function
```

3. 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 16 GB memory. The average run-time for different circuits are listed in Table 1. For small-scale and medium-scale circuits, our algorithm can reconstruct the circuit control logic from a flattened netlist in less than 1 min (less than 1 s in most cases). The run-time is below 10 min even for large-scale circuits. From Table 1, 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 [24] circuit finishes faster than the smaller RS232 transceiver due to the RSA circuit's more regular circuit structure.

Table 1
Average run-time for sample circuits.

Testing circuits	Registers	Total gates	Run-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

3.1. 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.

Using the flattened RS232 netlist as the input, our REFSM tools recover the control logic in the format of FSM of the entire circuit. Fig. 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.

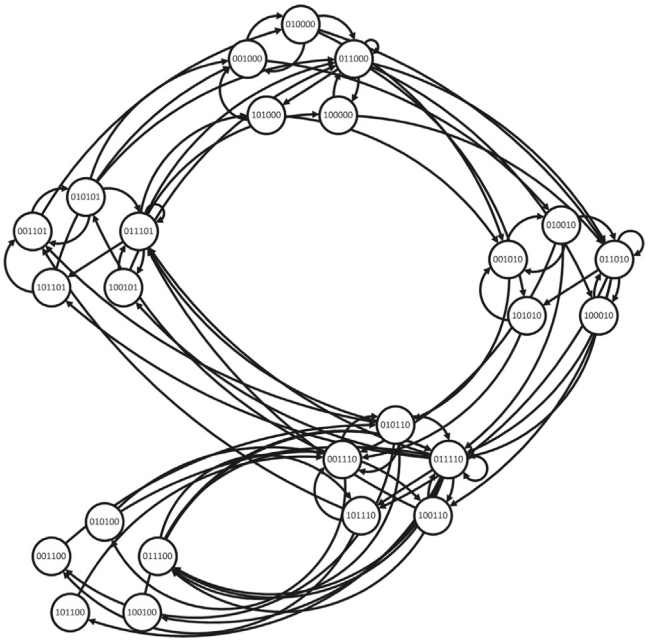
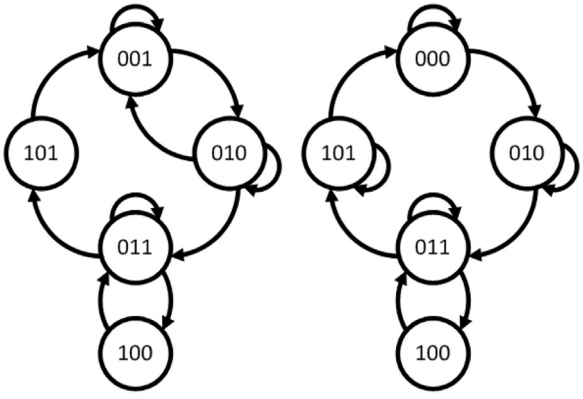


Fig. 2. Recovered control logic of the entire RS232 netlist.



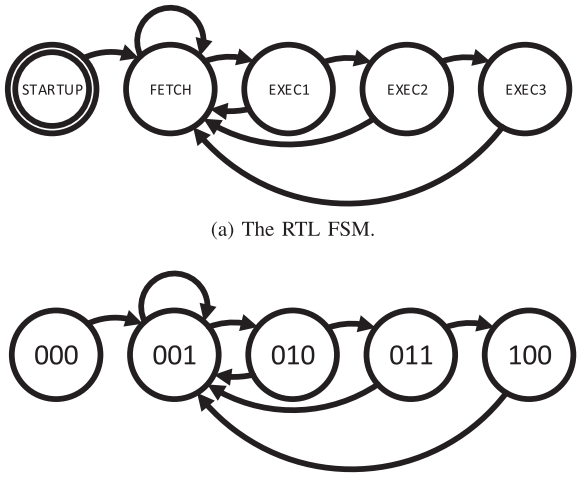
(a) First decomposed FSM. (b) Second decomposed FSM.

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

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

3.2. 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 [25]. Based on the RTL code, we first constructed the real FSM when dealing with different instructions (see Fig. 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 Fig. 4b. A comparison between Fig. 4a and 4b shows us that



(a) The RTL FSM.

(b) The REFSM FSM.

Fig. 4. The FSM recovered from MC8051 netlist and RTL (a) The RTL FSM, (b) The REFSM FSM.

these two FSMs are of the same structure. In fact, the transition conditions are also identical.

4. 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 [26]. 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 [27,28]. The platform is an FPGA implementation designed to perform all necessary operations for ciphertext 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 [28].

However, if we can identify all states of the Trojan FSM, determining 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. Fig. 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

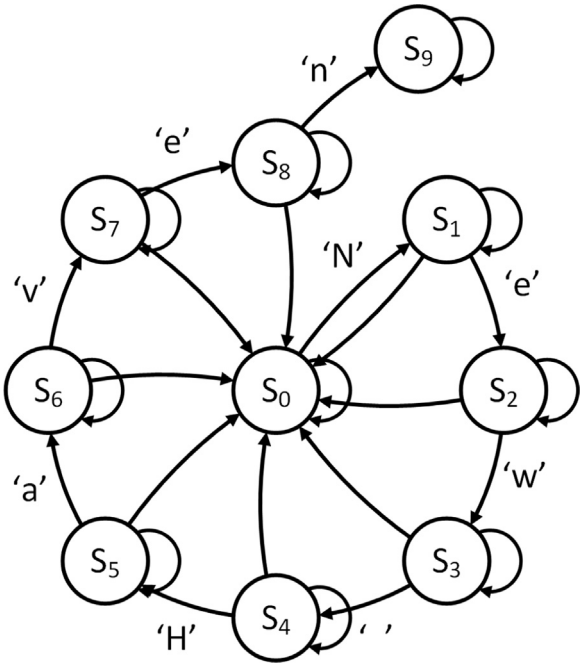


Fig. 5. The recovered hardware Trojan trigger.

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 further 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 [24]. Table 2 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.

5. REHOP for design obfuscation

Reverse Engineering Hardware Obfuscation Protocol (REHOP) inserts additional gates to a gate level netlist with the intent to prevent IP piracy. Previous methods for hardware protection via netlist obfuscation have been proposed [16]. Such methods create additional states within the netlist's logical FSM. These states comprise what is commonly referred to as the obfuscation mode, while the original states are called the normal mode. While the FSM is in obfuscation mode the modules of the circuit have a sizable chance of not performing properly, such that it becomes impossible to determine the actual function of the IP.

Previous methods do not scale well; the method of adding each new state caused the insertion of many gates, which is why re-synthesis was required. This limited the desired number of inserted states. The case studies covered in the previous methods can be bypassed by an intelligent adversary, who is without access to the gate level netlist. Determining the full FSM of a four register state machine with transitions conditions relying upon ten input signals requires no more than 16000 input vectors, assuming there is a scan chain [29].

5.1. Attack model

Attack models in previous work are often over-simplified. It has been assumed that adversaries behave randomly when stealing IP. That is, an attacker will only randomly select a subset of State Elements (SE or registers), a subset of primary input, a subset of output, and an input sequence to unlock the IP core [16]. This model is unrealistic and, therefore, it becomes difficult to measure the protection level of an obfuscation protocol using the attack model.

Instead, in this paper, we assume that an adversary has access to the scan chain of the netlist. Moreover due to the trial and error, the adversary has determined the SEs that belong to the FSM. This allows the adversary to, within a finite amount of time, determine the FSM of the netlist. This becomes the reason that one of REHOP's goal is to make full FSM recovery infeasible.

5.2. Obfuscated FSM

Similar to previous methods, REHOP creates additional FSM states by inserting a few SEs and other gates. The basic version of REHOP inserts two SEs that determine the FSM's mode. These two SEs control whether the FSM is one of 4 sub-modes, which can almost ensure that key guessing will lock the chip until it is reset. Unlike other methods the obfuscation mode is broken into 3 potential modes. The first mode is also called the standard obfuscation mode that can reach the normal mode with enough "correct" input, i.e., a particular subset of the primary input meets a specified requirement, which has been in the past determined by a PUF [16]. The second obfuscation mode is a penalized obfuscation mode. This mode is reached by inputting an incorrect sequence while in the standard obfuscation mode. With enough correct input sequences the FSM can reach the obfuscation mode. The second mode allows the user an additional chance to unlock the circuit by potentially re-entering the first mode. The third mode is the black hole mode. In this mode no other modes can be reached (including the normal mode). The chip needs to be reset to enter the function mode once in the blackhole mode. This mode can be reached by inputting incorrectly in the penalized obfuscation mode (the second mode). To bolster the size of each FSM a subset of the SEs already within the netlist are utilized to store the state of the FSM.

Table 2
Run-time and Trojan detection capability on Trust-Hub benchmark.

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

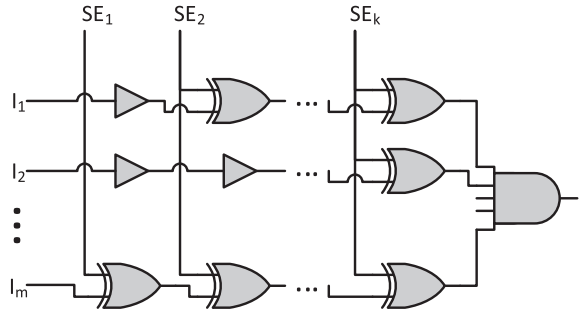


Fig. 6. An example of a key generating module used in REHOP.

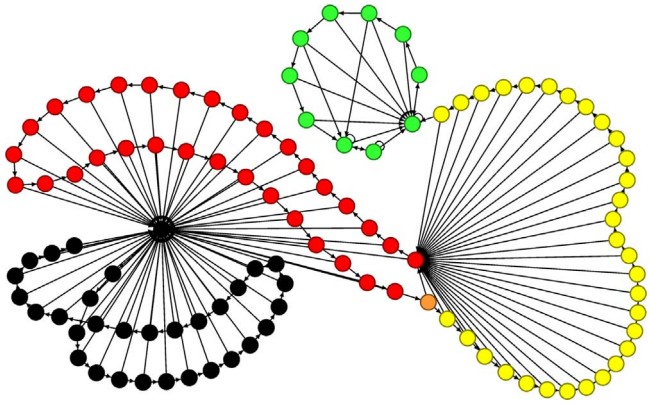


Fig. 7. A Function FSM (green nodes) with two obfuscated FSM (yellow, orange and red nodes) with a blackhole FSM (black nodes) combined into one netlist using REHOP. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5.3. Key generation

A number of methods can be employed for creating the "correct" sequence of keys. The latest version of the REHOP utilizes a look up table whose input comes from the SEs of the obfuscated FSM. An example of a look-up function could be XOR of random words dependent on the SEs that are logic-1. The output of this look up module is then XOR-ed with a subset of the primary inputs. The resulting set of values are put into an AND tree. If the result is logic-1, then the key is considered "correct". Otherwise the state will suffer some penalty, assuming the FSM is in some obfuscation mode. See Fig. 6 for an example gate-level implementation of the look-up table.

This method does not necessarily use the whole state space, that is there might be states unreachable from the FSM's start state. A knowledgeable attacker only needs to try at most $3(2^{|PI|})(2^{|SE|})$ to determine the correct sequence to unlock the Netlist, where PI is the chosen subset of primary inputs and SE is the set of chosen state elements that represents the obfuscated FSM. More complex (and protective) methods can be generated utilizing random particular combinations of SE as additional XORs.

5.4. Case study

To test our tool a simple FSM is run through our protocol. Fig. 7 shows the resulting graph representation of an FSM that has undergone the REHOP obfuscation process. The orange node represents the starting state. The yellow nodes represent the states leading to the normal FSM. States capable of reaching the black hole states with an incorrect key are represented by the red nodes. The nodes that act as the black hole states are colored black. Due to the small number of registers used in the original Netlist the FSM (normal FSM and the obfuscation FSM combined) is quite small. The number of test vectors required to recover this was around 60,000,000, largely because the number of inputs in the look up table was 20. Luckily as the number of registers increases the required number of test vectors for design recovery increases exponentially.

5.5. Discussions

The next step of REHOP is to improve itself to the point that the adversary model can be strengthened. Many goals needs to be met to achieve strengthen the adversary. First goal should be to improve the design for the look-up. Currently the degrees of freedom (DOF) of the look up table is the number of SEs used as input, that is the number of vectors that the adversary needs to learn is $|SE|$. A simple method for improving the DOF is to utilize combinations of SEs for updating the return value, but this model only increases the DOF by the number of inserted combinations. That is to utilize the full entropy of the system the size of the look up module would (with the proposed method) have to be exponential in size, which would defeat the purpose of the compact design. The saving grace of the original method is that if the gates of the table are camouflaged (which would need to be done in either scenario due to the vulnerability of the look up table) it takes an exponential amount of time to determine each word, with respect to the amount of inputs.

A second goal that should be reached to extend the adversary model to determine how to hide and complexify the FSM structure. If the adversary understands that the AND tree needs to be logic-1, then the IP pirate can easily extract the entire design by implementing the netlist in software with the AND tree connect to 1. Additional SEs might be inserted to allow the FSM to break to different realizable normal mode FSM. the drawback being that the size of the sequential logic increases which is undesirable due to its cost.

6. 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 reconstruction. 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. Supported by the REFSM tool, an automatic netlist security enhancing framework is also developed which can help obfuscate the control logic without referring to the RTL code. The new tool, REHOP, provides IP developers and IP distributors an effective solution to protect third-party IP, especially those in the format of gate-level netlist.

References

[1] M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, 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, S. King, Defeating UCI: Building stealthy and malicious hardware, in: in: Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP), 2011, pp. 64–77.

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

[4] E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: a pathway to trusted module acquisition, IEEE Trans. Inf. Forensics Secur. 7 (1) (2012) 25–40.

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

[6] Y. Jin, Y. Makris, A proof-carrying based framework for trusted microprocessor IP, in: Proceedings of the 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2013, pp. 824–829.

[7] Y. Jin, B. Yang, Y. Makris, Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing, in: Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2013, pp. 99–106.
[8] D. Lin, S. Eswaran, S. Kumar, E. Rentschler, 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, Y. Makris, Hardware Trojan detection using path delay fingerprint, in: Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust, 2008, pp. 51–57.
[10] C. Lamech, R. Rad, M. Tehranipoor, 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 Trans. Inf. Forensics Secur. 6 (3) (2011) 1170–1179.
[11] DARPA, Microsystems Technology Office, Integrity and reliability of integrated circuits (IRIS), 2010, dARPA-BAA-10-33.
[12] W. Li, Z. Wasson, S. Seshia, Reverse engineering circuits using behavioral pattern mining, in: Proceedings of the Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on, 2012, pp. 83–88.
[13] W. Li, Formal methods for reverse engineering gate-level netlists, Master's thesis, EECS Department, University of California, Berkeley, Dec 2013. [Online]. Available: ⟨http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-222.html⟩
[14] W. Li, A. Gascon, P. Subramanyan, W.Y. Tan, A. Tiwari, S. Malik, N. Shankar, S. Seshia, Wordrev: Finding word-level structures in a sea of bit-level gates, in: in: Proceedings of the Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, 2013, pp. 67–74.
[15] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W.Y. Tan, A. Tiwari, N. Shankar, S. Seshia, S. Malik, Reverse engineering digital circuits using structural and functional analyses, IEEE Trans. Emerg. Top. Comput. 2 (1) (2014) 63–80.
[16] R. Chakraborty, S. Bhunia, HARPOON: an obfuscation-based socdesign methodology for hardware protection, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 28 (10) (2009) 1493–1502.
[17] K.S. McElvain, Methods and apparatuses for automatic extraction of finite state machines, U.S. Patent 6 182 268, 2001.
[18] L. Yuan, G. Qu, T. Villa, A. Sangiovanni-Vincentelli, An fsm reengineering approach to sequential circuit synthesis by state splitting, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 27 (6) (2008) 1159–1164.
[19] Y. Shi, C. W. Ting, B.-H. Gwee, Y. Ren, A highly efficient method for extracting fsms from flattened gate-level netlist, in: Circuits and Systems (ISCAS), in: Proceedings of the 2010 IEEE International Symposium on, 2010, pp. 2610–2613.
[20] E.I. Goldberg, M.R. Prasad, R.K. Brayton, Using sat for combinational equivalence checking, Design, Automation and Test in Europe, 2001, pp.114–121, 2001.
[21] J.B. Wendt, M. Potkonjak, Hardware obfuscation using puf-based logic, in: Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, ser. ICCAD '14, 2014, pp. 270–277.
[22] B. Liu, B. Wang, Embedded reconfigurable logic for asic design obfuscation against supply chain attacks, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 201, 4, 2014, pp. 1–6.
[23] W. Imrich, Factor cardinal product graphs in polynomial time, Discret. Math. 192 (1–3) (1997) 119–144.
[24] ⟨https://www.trust-hub.org/⟩.
[25] Oregano Systems, 8051 IP core, ⟨http://www.oreganosystems.at/?Page_id=96.⟩
[26] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: Functional identification of gate-level hardware trustworthiness, in: Proceedings of the Design Automation Conference (DAC), 2014.
[27] ⟨http://isis.poly.edu/esc/2008/index.html⟩.
[28] Y. Jin, N. Kupp, Y. Makris, Experiences in hardware Trojan design and implementation, in: Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust, 2009, pp. 50–57.
[29] P. Subramanyan, S. Ray, S. Malik, Evaluating the security of logic encryption algorithms, in: Proceedings of the 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2015, pp. 137–143.