RERTL: Finite State Transducer Logic Recovery at Register Transfer Level

Jason Portillo*, Travis Meade*, John Hacker*, Shaojie Zhang*, Yier Jin†
*Computer Science department, University of Central Florida
†Electrical and Computer Engineering department, University of Florida
jasonlportillo@gmail.com, travis.meade@ucf.edu, jhacker997@knights.ucf.edu, shzhang@cs.ucf.edu, yier.jin@ece.ufl.edu

***Abstract*—Increasingly complex Intellectual Property (IP) design, coupled with shorter Time-To-Market (TTM), breeds flaws at various levels of the Integrated Circuit (IC) production. With access to IPs at all stages of production, design defects can easily be found and corrected, i.e., knowledge of the Register Transfer Level (RTL) code allows for the option of easy defect detection. However, third-party IPs are typically delivered as hard IPs or gate-level netlists, which complicates the defect detection process. The inaccessibility of source RTL code and the lack of RTL recovery tools make the task of finding high-level security flaws in logic intractable. Upon this request, in this paper, we present an RTL recovery tool suite named RERTL that leverages advanced graph algorithms including Lengauer-Tarjan's dominator tree and Euler tour tree technique to assist in netlist analysis. Supported by RERTL, logical states and their interactions are recovered from the initial design in the format of gate-level netlists. After the recovery of state interaction, RERTL further converts the full design into human-readable RTL. A series of netlist case studies were examined using RERTL covering benign logic structures, designs with accidental defects, and designs with deliberate backdoors. The experimental results show that all of our designs at various complexities were recoverable within seconds.**

I. Introduction

Backdoors, watermarking, Third Party Intellectual Property (3PIP), and malicious logic insertion, or at least their potential, creates a paranoia suffocating the integrated circuit industry. All have contributed to a wave of hyper awareness of the need for contract fulfilling modules. In severe cases, out-of-specification functionality can devastate complex, computational systems that rely on 3PIPs to abide by predefined contracts. Finding modules that violate such contracts can be especially tricky, if such undocumented functions rarely occurs, such as the case with Hardware Trojans. Intentionally malicious designs have been discussed at length in research [1]. A passive stance on the issue is unsound, since the occurrence of such behavior in the field might be improperly attributed to engineering or fabrication flaws. Devices already exist that are at risk of containing such logical defects. To protect developers, researchers have aimed to prove high level circuit functions or generating input patterns that theoretically cover all possible netlist function. However, such practices typically rely on the knowledge of certain high-level descriptions that 3PIP developers could be reluctant to distribute. Developers are concerned about piracy as much as consumers are concerned with reliability, and since developers control how IP is distributed, consumers are left with the burden of using compromised IP or developing a full system.

978-1-7281-3544-1/19/$31.00

Methods that are designed to formally verify netlist can require a detailed description of functionality whose construction is harder to obtain than actually creating the design itself. Because of the large resources required in formal analysis, it is easier for developers to extract the high-level description of a RTL design which can be measured in thousands of lines of hardware description language (HDL) code, compared to a netlist with millions of gates. Without access to the original design, some methods must be developed that can assist developers in understanding the high-level description of a netlist. Large netlists are composed of logic and memory components. The memory is repetitive and undergoes the same basic operations based on the logical state. The logic takes up a large portion of netlists and dictates the high-level function of such designs. For these reasons RTL verification methods can only be leveraged on gate-level netlist when users are able to translate the design's logic to the high-level description. This paper expands upon the existing tools that can be used to extract high-level circuit descriptions from low-level netlists.

However, with the ability to extract the function of a netlist comes the ability to reproduce, modify, and redistribute IP. As long as modifications are not malicious to the community of 3PIP, users do not need to worry too much, but IP developers need to be aware of this issue. Material that IP providers have been willingly sharing with others with the assumption that gate-level or even FPGA Bitstream is too difficult to pirate, is actually at risk of being stolen.

This paper is, to the best of our knowledge, the first to approach the netlist verification problem with the presented model (i.e. netlist to partial RTL for verification). The topic and methods discussed create many new interesting problems for IP verification and reverse engineering for hardware security.

This paper is structured in the following format. Section II focuses on methods that have been proposed and analyzed in previously published works. In Section III, we go into depth discussing the issues with current methods and the two general sub-problems we faced in converting and creating tools for RTL generation. Then, Section IV discusses in detail how the two problems are solved in addition to some alternative

equivalent schemes. After that, Section V discusses how we analyzed our method and includes the results of the analysis. Finally, Section VI discusses at a high-level our findings and possible modifications and improvements.

II. Related Works

A. Hardware Trojan Detection

One security concern for IP consumers is that of an intentionally afflicted design. Malicious circuitry purposefully embedded in ICs are referred to as Hardware Trojans, and many methods exist to assist in maintaining a Trojan free production line. A machine learning based method for detection went so far as to use SVMs to analyze metal layouts for significant discrepancies, to reduce the time humans would need to spend looking for behavior altering circuitry [2]. This method although highly effective, required getting the correct low level layout of the original netlist. An early method for design verification is Trojan free was side-channel analysis. Information such as path delay has previously been used [3]. Many methods still require the knowledge of the original netlist. Other side channel methods have been proposed that do not rely on original netlist information [4]. Instead such methods assume that the Trojan will be inserted in a design post place and route to ensure path delays are increased in such logic. Unlike former method the latter only requires the original side-channel information, rather than the full design. 3PIP developers are more inclined to share side channel information over high-level specifications. However, if the 3PIP developer is unavailable or malicious, such methods will fail. Neither of these solutions are guaranteed to work or be available when using a hard IP core delivered by a 3PIP vendor.

In terms of looking into the gate-level netlist itself, hardware Trojan detection methods have been proposed [5]. Typically these methods look for bits that are stuck at values or determine where sensitive pieces of information is capable of leaking. However, Trojans that do not rely on specifically high-low trigger signals and Trojans that target denial of service (DoS) can bypass such methods. In general, methods that automatically determine Trojan insertion on behalf of the user is prone to error, and methods that use machine learning can be poisoned. More general netlist annotation/Reverse Engineering based methods are desirable.

B. Netlist Reverse Engineering

Third party resources will not always offer high-level descriptions. Tools exist to help annotate and analyze netlists. A framework HAL gives users the ability to graphically see a netlist's topology [6], [7]. Additionally, HAL can allow for python plugins to automatically run external algorithms on such designs. However, such tools require users to have an extensive knowledge or third party resources consisting of netlist analysis algorithms. Such algorithms will be discussed in further detail within this sub-section.

Methods that do not leverage high-level IC access use Reverse Engineering techniques to look for any potential problems. One method for high-level component recovery was proposed in [8]. The method cut the netlist into segments, identified words, found modules, and looked up functions. The main issue such a method can have is the lack of identifiable circuit modules. Missing modules can be the result of learning the exact behavior of some synthesizer, but using a different synthesizer when constructing the netlist can easily leave sections of the netlist unclassified. Most likely a 3PIP user will not choose from which synthesis tool their design originates. Methods have shown that data paths can be reconstructed, which can be useful when a Trojan might divert sensitive signals within an integrated circuit [9]. When and where data flows is largely missing from such designs, since these methods do not explicitly analyze the high-level control signals.

Fortunately, methods have been proposed to reduce the low gate-level logic into a higher form, such as a logical Finite State Machine (FSM) [10]–[13]. As an example, one previous method for extracting high-level descriptions from low-level netlists is called REFSM [10]. The method recovers the logic of a gate-level netlist in the form of an FSM, but the information contained within the logic appears to be solely the FSM component. These Reverse Engineering methods could be implemented on top of HAL's framework. However, such fusion of tools would required re-implementing many existing tools into one of HAL's plugins.

Nevertheless, FSMs are not the easiest thing to read and, depending on the design, can be much larger than the initial RTL. RTL, a format for representing circuit logic, already exists and is universally accepted among IC/IP designers. Few tools exist to analyze an FSM for potential flaws, but many tools for analyzing RTL exist. For these reasons, a more robust description with RTL is of higher quality in terms of assisting in securing hardware. Works have demonstrated that logical RTL can be represented by an FSM [14]. The RTL at a high level contains just states, transitions, and outputs. This previous work showed a method for conversion of the RTL to an FSM, but the reverse problem has largely been ignored. For this reason we present a method that extracts the RTL from the FSM. In addition, our proposed method extracts the signals emitted from the FSM to help determine, if certain states might be watermark or malicious states inserted out of specification.

III. Motivation and Attack Model

This paper focuses on hardware security by converting low-level circuit descriptions into higher level formats. Methods in the past have converted gate-level netlist descriptions into an FSM [10], [11], but depending on the auxiliary information presented with the FSM or even the structure of the FSM, determining the reliability of a netlist is still challenging. This is why we focus on recovery of the netlist's logic in the form of a simplified snippet of RTL code with extra information pertaining to the signals controlled by the FSM. Developers that design circuits can more easily interpret the RTL than an FSM, especially developers familiar

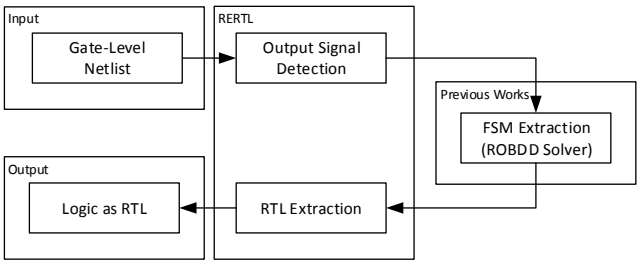

Fig. 1. Overview of a netlist reverse engineering flow and the proposed RERTL tool.

with different designs and common errors. Many tools exist that formally analyze RTL than FSMs. Thus RTL could allow for quick, accurate fault detection over an FSM description, regardless of the analysis required. Moreover our method is capable of capturing the signals that can control the flow of data in the netlist. For this reason we occupy much of our paper with extracting the output from the FSM. An FSM with the output information is conventionally called a Finite State Transducer (FST).

Our method assumes that we have access to a gate-level description of the circuit and basic knowledge of logical components, such as logical signals, primary inputs, and primary outputs of the design. To discover logic-data interaction we find fanout signals of the logical state elements which interact with other input or data signals. After identifying these signals we can determine more clearly how the state machine interacts with the input. After acquiring all the transitions and the emissions, the resulting data may be too large to manually check. Even without state machine outputs FSMs can be several megabytes in size. For this reason we implemented methods to convert the generated FST into RTL. Conditions extended by the output and states might be made overly complex compared to the original design. Managing the large function of state crossed with inputs to generate particular outputs requires a careful touch to avoid sluggish reduction methods. In the end we partially utilize previously described methods for extracting higher-level descriptions as seen by Figure 1. The problem of simplifying the results to a human-readable format is especially challenging when faced with the problems of 1) State machine output finding and 2) RTL formulation post state machine extraction. In fact, the main contributions of RERTL and this paper are our solutions to these two problems.

IV. Methodology

In this section, we discuss our methods used for analyzing gate-level netlists and generating high-level RTL. Our tool, RERTL, can be divided into three major steps.

- First, a pre-process will be performed to find all output signals of a logical word's FSM.
- second, a modified REFSM tool will be leveraged to recover the state and transition logic as well as the outputs signals evaluated at each state.
- Finally, the transitions are analyzed, conditions are reduced, and the resulting RTL is produced.

A. Finding the Outputs of the FST

A major step toward RTL recovery involves determining which signals are most likely the "output" of the FST. There could be a large amount of signals leaving the FST, and the combinational circuit alone proceeding the state registers could consist of thousands of gates. Many signals which rely on the state of the transducer might not even affect data directly but could indirectly modify data via the combination with other input signals. A goal of output wire detection is to find the farthest signals that affect signals outside of the FST's scope but are exclusively controlled by the state registers. The frontier should be composed of the signals that interact with logic directly outside the FST's output signals.

Algorithm 1 Returns a set of output signals for a given Logic word, W, and the netlist, N, as a signal graph

```
1: function FindOutput(W, N)
2:     N' is the reverse signal graph of N
3:     N_s is a signal graph where registers in N are split into input/output nodes
4:     N'_s is a signal graph where registers in N' are split into input/output nodes
5:     Add a root and FSTroot node to N_s
6:     Add a root and FSTroot node to N'_s
7:     for r ∈ register set of N do
8:         if r is from W then
9:             Add Edge from FSTroot to r's output Node in N_s
10:            Add Edge from FSTroot to r's input Node in N'_s
11:        else
12:            Add Edge from root to r's output Node in N_s
13:            Add Edge from root to r's input Node in N'_s
14:        end if
15:    end for
16:    Connect root to FSTroot in both N_s and N'_s
17:    DT is the Dominator Tree of N_s where the root is root
18:    DT' is the Dominator Tree of N'_s where the root is root
19:    BFS over nodes dominated by FSTroot in DT
20:    output is an empty set that will contain the output signals
21:    for u ∈ BFS do
22:        for x in u's fanout do
23:            Determine reachability of x via Euler Tour Technique on DT
24:            if x not dominated by FSTroot in either DT/DT' then
25:                Add u to output
26:            end if
27:        end for
28:    end for
29:    return output
30: end function
```

Finding signals that are exclusively controlled by the logical registers can be thought of as finding signals that are "dominated" [15] by the logic registers. We could "merge" the state logical registers and find dominated signals. Thus the problem is reduced to finding the subtree in a dominator tree. Algorithm 1 presents a high level implementation of output signal detection by leveraging dominator trees. We create an alternative graph based on the netlist where two extra roots are added, the main root and an FST root. The FST root will dominate every output and frontier output signal leaving the state registers. To extract the frontier we have to find all dominated signals that are either netlist output signals or fan-

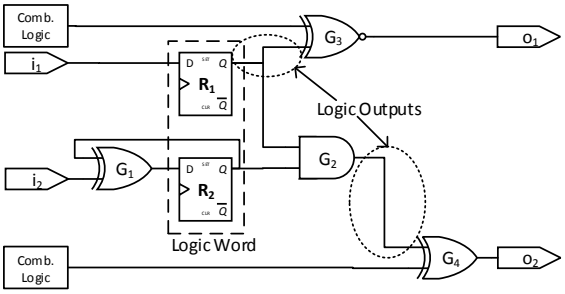

Fig. 2. Example gate-level netlist with circled output signals.

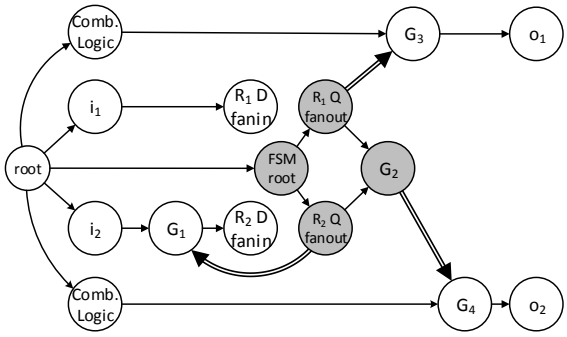

Fig. 3. Forward graph of the algorithm. The shaded nodes are dominated by the FSM root. The double arrows show potential FSM output signals.

out directly to a non-dominated signal. To do this, we look at the wires in BFS order starting at the FST root, and for each edge check if the next wire is no longer dominated. However, this algorithm can find false-positive output signals that only feedback into the state registers. Therefore, we have to run the entire algorithm again in parallel, except reversing the direction of all edges and moving the location of both roots to the outputs instead of the inputs. Doing this will allow us to eliminate signals that are completely dominated by the state registers in the reverse direction as seen on Lines 4, 6, 10, 13, 16, and 18 in Algorithm 1. Since dominator trees can be found in Linear time, the run-time for this method will take no more than O($|N|$ + $|E|$) time, where $|E|$ in this case is the sum of the fan-in sizes (usually no more than $6\times|N|$) plus the number of state registers plus 1.

For example, in the netlist shown in Figure 2, we have a simple state transducer that has two output signals affecting logic outside of its scope. Figure 3 is the constructed forward graph which we will run Lengauer-Tarjan's dominator tree algorithm on. The shaded nodes are directly or indirectly dominated by the FSM root. All wires leaving these nodes are potential output signals of the FSM. However, as we can see in Figure 2, the wire leaving R_2 and entering G_1 is contained entirely within the state machine. We can use the reverse graph in Figure 4 to eliminate this signal that is dominated by the FST root in the reverse direction. Improper dominator tree usage would have flagged the second register's output as an output signal. But due to the reverse graph check, we find that only two signals, Gate 2 and register 1, are actually

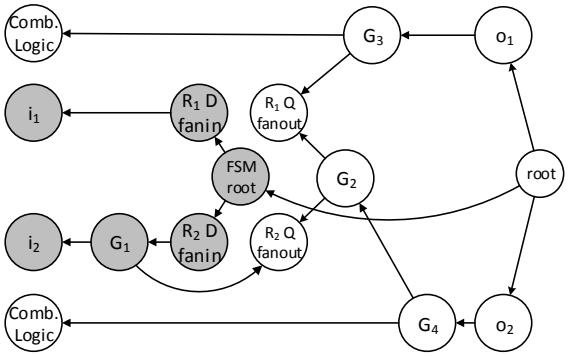

Fig. 4. Reverse graph of the algorithm. The shaded nodes are dominated by the FSM root.

logical output signals.

B. RTL Code Generation

A Reduced Order Binary Decision Diagram (ROBDD) solver is leveraged to extract the raw FST from the netlist, and we will transform the data into a more human-readable format described in RTL code. Given the resulting transitions and a list of inputs, RERTL can generate high-level RTL that is equivalent to the recovered FSM. To guarantee any FST's representation, we chose a general output form for our RTL code which includes a header, a body and a reset logic. The header defines all the wires used for input, output, and states and how they relate to the wires in the netlist. The body is used to define all of the reachable states as well as the logic for each transition. The footer is used to define the reset state and applying the transition chosen in the body. The remainder of this section will focus on describing the detailed process of generating RTL code in this form.

1) Output Wire Reduction: The first step of RERTL is to analyze the outputs of each state given by REFSM. Our method described in Section IV-A for finding output wires is very lenient, meaning that the final set of wires is typically a superset of the actual output wires. To reduce the set of wires, we will remove any wires that are constant across all states or strict negations of a wire already in the selected wire set.

2) Generating Conditions: The next step to recover the RTL is generating the conditions for each state transition. Given a single input word with n bits, there can be d don't-care bits and $n-d$ bits critical to the state transition in a single transition. Furthermore, the input word can be prefixed in the least significant bits with p don't-care signals, where $p \leq d$. The following four cases are all the possible ways to describe a transition given by the ROBDD solver:

- Case 1: $d = 0$
- Case 2: $d = p$
- Case 3: A brute force method could compute 2^{d-p} in a reasonable time
- Case 4: Brute force is not sufficient for an acceptable time constraint

Case 1 is the trivial case where all of the bits are critical and defined in this transition. This can be represented as a simple equality check. In Case 2, p is important because all don't-care signals are a prefix of the input word and can be compressed to represent a single continuous range. In Case 3, these p bits can be removed and compressed in the same way. This allows us to use brute force by checking all possible combinations of the remaining $d - p$ bits and generate all ranges that satisfy this condition. In Case 4, the computational time for brute force would be too expensive. Moreover, the number of conditions would be too large to fit into one IF statement. Instead, we can use bitwise AND logic to represent the transition very concisely. As a trade-off, we lose some of the reasoning and logic behind the transition.

The following examples represent each mentioned case and the condition that is generated, where "X" in the input word represents a don't-care signal:

```
Case 1: 000101 → word == 6'd5
Case 2: 0001XX → word >= 6'd4 && word <= 6'd7
Case 3: 00X1XX → (word >= 6'd4 && word <= 6'd7) ||
        (word >= 6'd12 && word <= 6'd15)
Case 4: X0X1XX → (word & 6'b010100) == 6'b000100
```

Once all of the transitions for a state are generated, we sort the transitions by the number of conditions associated with it. This is done to reduce complexity and take advantage of the if-elseif-else scheme of the transition. Since they are sorted, the most complex condition will appear last, resulting in being replaced with a simple else statement.

V. Benchmarks and Results

We looked into a number of ISCAS benchmarks [16]–[18] and designs from the Trust-Hub [19], [20]. Many of the designs used have explicit state machines while other have implicit (or no) state machines. We treat the design with implicit or no state machine as one state circuit. In most cases we selected the most obvious submodule for recovery. In one case in particular, the S13207, two different logical submodules were tested independently.

All of our designs were run on a computer with 16GB of memory and an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz. The first two parts of RERTL, output finding and the modified REFSM, were implemented in C++, while the third part was implemented in Java.

Verification was performed manually on many of the small designs. The larger designs were checked by comparing RERTL's FST to the conditions generated by REFSM. The RTL of the smaller Trojan infected designs was compared with the original design to validate the correct recovery. RERTL has the ability to change the order of the input bits to allow for easy verification. The output associated with the states was checked in the designs that had original RTL for comparison.

Table I shows the results of RERTL ran on various gate-level netlists. The first column shows the name of the design. The second column shows how many gates are in the design. The third and fourth column describes the size of the FSM in terms of the number of states and the number of transitions. The amount of time taken to finish for the whole tool compared with the time taken on REFSM is shown in columns 5 and 6, respectively. The RERTL run time shown is the time taken to find the output signals, extract the transitions, and convert into FST. The second to last column of the table is used to show the size difference between the recovered RTL and FSM as a ratio of the RTL size in bytes over FSM size in bytes. The last column shows that the design recovered by RERTL emulates an FSM identical to the one found by REFSM.

The first example we will cover demonstrates RERTL's ability to identify and recover a Trojan injected into an AES circuit. Listing 1 shows the original RTL of a Trojan triggering based on a specific passcode received through input. Listing 2 shows the RTL recovered by RERTL. There is only one trivial difference between the two, the recovered version's output is a negated version of the Tj_Trig signal.

Listing 1
Original AES T1000 Trojan Trigger RTL Trigger

```
always @(rst, state)
begin
  if (rst == 1) begin
    Tj_Trig <= 0;
  end else if (state == 128'
      h00112233_44556677_8899aabb_ccddeeff)
      begin
    Tj_Trig <= 1;
  end
end
```

Listing 2
Recovered AES T1000 Trojan Trigger RTL

```
assign inputs = {state[127], ..., state[0]};
assign en = {Tj_Trig_Bar};
always@(*)
begin
  case (curr_state)
    1'd0 :
      begin
        en <= 1'b1;
        if((inputs == 128'h112233445566778899
            aabbccddeeff))
          next_state <= 1'd1;
        else
          next_state <= 1'd0;
      end
    1'd1 :
      begin
        en <= 1'b0;
        next_state <= 1'd1;
      end
  endcase
end
always@(posedge clk, posedge rst)
begin
  if(rst)
    curr_state <= 1'd0;
  else
    curr_state <= next_state;
end
```

VI. Conclusion

In this paper, we have discussed a motivation for having high-level netlist descriptions for analyzing IP. RTL was

TABLE I
THE RESULTS OF RERTL ON VARIOUS NETLISTS

Netlist	Gates	States in FSM	Transitions in FSM	RERTL Time	REFSM Time	RTL/FSM Size Ratio
b1	24	8	25	1.28s	0.03s	1.31
b2	144	7	10	1.28s	0.04s	3.53
b4	632	3	3	1.23s	0.04s	6.38
b5	821	68	71	1.26s	0.04s	4.17
b7	380	43	43	1.25s	0.04s	3.05
b9	146	262401	459009	27.18s	3.11s	1.53
b10	174	611	3711	1.75s	0.08s	0.81
b13	310	17042	19182	2.95s	0.22s	1.44
S510	179	47	76	1.24s	0.03s	2.92
S641	107	1325	179185	8.56s	1.19s	0.75
S1423	490	4	12	1.24s	0.04s	1.91
S1488	550	48	168	1.31s	0.04s	1.50
S1494	558	48	168	1.29s	0.04s	1.87
S9234	2027	6	36	1.25s	0.04s	0.51
S13207 FSM 1	2573	64	127	1.30s	0.05s	2.08
S13207 FSM 2	2573	32	33	1.25s	0.05s	4.11
S15850	3448	4	12	1.26s	0.05s	1.25
AES T100 Counter	11175	1048576	2097151	114s	12.9s	1.37
AES T400 Trigger	11686	2	262	1.26s	0.06s	**0.00679**
AES T1000 Trigger	11269	2	272	1.43s	0.06s	**0.0193**

chosen as the method for representation over an FSM due to its intuitive description design. We have presented a novel method, RERTL, for RTL extraction. The proposed method was demonstrated on a set of netlists. The experimental results show that all of the benchmark designs at various complexities were recoverable within seconds. The future work is to improve the transition compression so that even larger complexity FSMs can be recovered. Additionally, the cases for transition compression can be extended, such that more common logic patterns, e.g., add, multiply, increment, etc., can be identified and reduced.

Acknowledgement

The work is partially supported by the National Science Foundation (NSF-1812071). Jason Portillo and John Hacker are supported by the REU Supplement of this project.

References

[1] S. Bhasin and F. Regazzoni, "A survey on hardware trojan detection techniques," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2021–2024.

[2] C. Bao, D. Forte, and A. Srivastava, "On application of one-class svm to reverse engineering-based hardware trojan detection," in *Fifteenth International Symposium on Quality Electronic Design*. IEEE, 2014, pp. 47–54.

[3] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *2008 IEEE International workshop on hardware-oriented security and trust*. IEEE, 2008, pp. 51–57.

[4] A. Amelian and S. E. Borujeni, "A side-channel analysis for hardware trojan detection based on path delay measurement," *Journal of Circuits, Systems and Computers*, vol. 27, no. 09, p. 1850138, 2018.

[5] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*. IEEE, 2017, pp. 1–10.

[6] M. Fyrbiak, S. Wallat, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, R. Tessier, and C. Paar, "HAL- the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[7] EmSec Chair for Embedded Security, "HAL - The Hardware Analyzer," https://github.com/emsec/hal, 2019.

[8] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 1277–1280.

[9] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "Wordrev: Finding word-level structures in a sea of bit-level gates," in *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE, 2013, pp. 67–74.

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

[11] M. Fyrbiak, S. Wallat, J. Déchelotte, N. Albartus, S. Böcker, R. Tessier, and C. Paar, "On the difficulty of fsm-based hardware obfuscation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 293–330, 2018.

[12] T. Meade, J. Portillo, S. Zhang, and Y. Jin, "Neta: when ip fails, secrets leak," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 90–95.

[13] T. Meade, S. Zhang, and Y. Jin, "Ip protection through gate-level netlist security enhancement," *Integration*, vol. 58, pp. 563–570, 2017.

[14] C.-N. Liu and J.-Y. Jou, "A fsm extractor for hdl description at rtl level," in *Proc. of Asia-Pacific Conference on Hardware Description Languages*, 1998, pp. 33–38.

[15] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, 1979.

[16] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *Proceedings of IEEE Int'l Symposium Circuits and Systems (ISCAS 85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[17] F. Corno, M. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *Design Test of Computers, IEEE*, vol. 17, no. 3, pp. 44–53, Jul 2000.

[18] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Circuits and Systems, 1989., IEEE International Symposium on*, May 1989, pp. 1929–1934 vol.3.

[19] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st international conference on computer design (ICCD)*. IEEE, 2013, pp. 471–474.

[20] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, 2017.