NETA: When IP Fails, Secrets Leak

Travis Meade
Department of Computer Science
University of Central Florida
Orlando, Florida
travis.meade@ucf.edu

Jason Portillo
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, Florida
jasonlportillo@knights.ucf.edu

Shaojie Zhang
Department of Computer Science
University of Central Florida
Orlando, Florida
shzhang@cs.ucf.edu

Yier Jin
Department of Electrical and Computer Engineering
University of Florida
Gainesville, Florida
yier.jin@ece.ufl.edu

ABSTRACT

Assuring the quality and the trustworthiness of third party resources has been a hard problem to tackle. Researchers have shown that analyzing Integrated Circuits (IC), without the aid of golden models, is challenging. In this paper we discuss a toolset, NETA, designed to aid IP users in assuring the confidentiality, integrity, and accessibility of their IC or third party IP core. The discussed toolset gives access to a slew of gate-level analysis tools, many of which are heuristic-based, for the purposes of extracting high-level circuit design information. NETA majorly comprises the following tools: RELIC, REBUS, REPCA, REFSM, and REPATH.

The first step involved in netlist analysis falls to signal classification. RELIC uses a heuristic based fan-in structure matcher to determine the uniqueness of each signal in the netlist. REBUS finds word groups by leveraging the data bus in the netlist in conjunction with RELIC's signal comparison through heuristic verification of input structures. REPCA on the other hand tries to improve upon the standard bruteforce RELIC comparison by leveraging the data analysis technique of PCA and a sparse RELIC analysis on all signals. Given a netlist and a set of registers, REFSM reconstructs the logic which represents the behavior of a particular register set over the course of the operation of a given netlist. REFSM has been shown useful for examining register interaction at a higher level. REPATH, similar to REFSM, finds a series of input patterns which forces a logical FSM initialize with some reset state into a state specified by the user. Finally, REFSM 2 is introduced to utilizes linear time precomputation to improve the original REFSM.

ACM Reference Format:
Travis Meade, Jason Portillo, Shaojie Zhang, and Yier Jin. 2019. NETA: When IP Fails, Secrets Leak. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3288739

1 INTRODUCTION

Tariffs, high volume demand, low time-to-market, and first-rate chips requirements: all these factors block and sometimes directly oppose each other. Cracking and straining against each other, the forces driving the unstable market of Integrated Circuits (ICs) leaves many in a quandary, and such issues, if neglected, will further damage relations between Intellectual Property (IP) developers and users.

Recently an article slammed Apple in regards to hiding discovered hardware manipulations that could have lead to large scale sensitive information leaks [13]. The damning exposé almost immediately came under fire from Apple and Amazon for presenting misleading and false claims, of which the most important was that of the existence or knowledge of malicious hardware manipulations. Whether it was a valid recounting of terrors within the Hardware Security domain or potentially a fake news piece in a sensational hungry world from "anonymous" sources, the Bloomberg article refocused the hardware security community onto a question among many IP developers and users: "What is in my silicon?"

Among all possible answers, the technique of reverse engineering is one of the oldest but still the most reliable solution. With the ability to fully recover designs, consumers can verify for themselves the integrity of their products. The problems in this answer is the overwhelming research opposing such techniques, either by trying to demonstrate the ability to bypass Hardware Trojan defenses, or by trying to prevent piracy of IP.

In this paper we seek to help circumvent many trust issues that plague current IC development, by discussing a toolset for Netlist Analysis Toolset (NETA) [11] that leverages Reverse Engineering approaches to check netlists for potential malicious logic. We talk about other approaches for evaluating ICs in section 2. We give an overview and use of our current project in section 3, and show how not only can the proposed toolset help in assessing netlists, but can find rare triggering logic, by decrypting locked netlists. Section 4 presents our most recent developments in the NETA project. Finally, section 5 concludes the paper and presents future works regarding netlist analysis.

2 BACKGROUND

Methods that analyze circuits exist for many different phases and design descriptions. Researchers usually try to recover the chips layout information via the process of decapping and imaging. More specifically, layout information of the design for each layer can be generated by etching the surface of a chip, while imaging can be completed with the aid of a Scanning Electron Microscope (SEM) or using High Resolution X-Ray Tomography. After extracting the layout the gate level information is usually recovered using computer vision algorithms that tries to identify the cells and VIAs [19].

A plethora of methods have been proposed that analyze the gate-level netlist. A number of methods use netlist extraction and/or identification [1, 4, 14, 16, 18, 19]. Many of these methods work on the graph model of the design and typically include graph structure analysis. Analysis of netlist components has been thoroughly discussed and examined in [2, 6, 8, 16, 18]. Due to the nature of signal propagation the design of a netlist can be closely represented using a digraph, so many methods that can analyze graph structures can be re-purposed for gate-level netlist analysis.

Like the process of creating the chip itself, which involves many steps, algorithms, and tools, the process of recovery should allow for different recovery outcomes. Intermediate steps should exist whose results are observable and editable, in the event that the tool user has extraneous high-level information that could better improve the toolset's results. For this reason our toolset has many different functions, each of which is modularized in a way such that different netlist descriptions/levels of interpretation can be used without having to reduce the netlist down to simpler terms. The process of handling higher level descriptions is especially useful.

3 NETA TOOLSET

The first goal of the tool set is to help IP users analyze netlist for malicious code. To aid users in this endeavor the developers of the toolset tried to reduce the problem to that of high-level function recovery of netlists. This strategy allows us to off load part of the burden of malicious logic detection to the netlist user.

3.1 RELIC, REFSM, and REPATH

The first step to most netlist analysis techniques should fall to either netlist partitioning or logic identification. Knowing the logical components of a netlist is useful since it can allow for other analysis methods. RELIC 1 [7] uses a heuristic based fan-in structure matching to determine the uniqueness of each signal in the netlist.

Conceptually logic wires would have structure that is unique, while data wires should have a fan-in structure similar to the fan-in structures of the other wires in the same word. This conjecture allows for the base of a method for finding logic wires. RELIC 1 tries to match the wires composing the fan-in of a given gate pair such that the sum of the scores of the matched wires is maximized. To prevent infinite recursion a depth is used to cut off the search.

A second tool produced to help find logical wires, RELIC 2 [11], not only uses RELIC 1's method for finding structure similarity, but it also uses certain deeper fan-in and fan-out information to improve our toolset's capability at finding outlying wire structure. RELIC 2 merges the RELIC 1 information and deep wire information using Principle Component Analysis (PCA) and clustering techniques to flag suspicious signals.

For actual function recovery REFSM is typically relied upon [9, 10, 12]. Its goal is given a netlist and a set of registers, construct a FSM that represents the behavior of the register set over the course of the operation of the netlist. REFSM can be very useful for checking how register values interact with each other at a higher level.

A method similar to REFSM is that of REPATH. What REPATH tries to do at a high level is to find a series of input patterns that will force an FSM that starts at some reset state into a state that is specified by the user. The state can be singular or a set of states. The REPATH will always find the least number of steps needed to get to that state. However, since a user might not omit registers from the FSM word certain transition conditions might not be achievable at certain clock cycles used by the REPATH method.

3.2 Other Tools

One of the most popular tools used by researchers for finding the high level function behavior embedded in the gates of a circuit is that of Strongly Connected Component (SCC) finding. An SCC is a subgraph of a directed graph such that each node within the SCC is capable of reaching each other node. An FSM that leaves its state graph SCC will have effectively cutoff other FSM states. Transitioning out of an SCC implies that the control logic of the netlist has entered into a particular mode that effectively locks the behavior of the IC. This analysis can be used for finding "black hole" states within an obfuscated FSM or finding functioning states in a sequential locking FSM. The analysis also allows for a more detailed categorization of register and their behaviors and interactions, which can help identify FSM words [14].

For doing simple analysis on the register interaction the Register Dependency (REDPEN) tool can be utilized. The purpose of the REDPEN is to find the register dependency graph. The tool outputs the dependency as a transition graph, which can be used with Tarjan's SCC algorithm [17] for extracting groups of registers that can affect each other.

3.3 Case Study

One case study we performed took a netlist locked by several registers. We executed one of our standard analysis procedures, which can be seen in Figure 1.

The RELIC 2 tool was used to find registers that were most likely the logic registers of the controller for the system. RELIC 2 returns a list of estimated z-scores that represent how likely each wire is data register (compared to logic register). The least likely data wires within the netlist had z-scores of 23.31, 31.94, 105.45, 114.14, and 193.09, all of which are very unlikely. In the original case study we decided to try to extract a word from the wire that scored the 193.09, although the other wires could also be examined for potential controller behavior. In most analysis we try to keep the word as small as possible to enable quick word checking. When we ran the netlist through our SCC using REDPEN, we found a total of nine other signals that could interact with the 193.09 signal. Since ten registers can create fairly meaningful FSMs, we moved on to REFSM for logic extraction. One interesting result was that there was no reset signal in the circuit. It is an indication that the original circuit is not well prepared. If an FSM enters a black hole state, then the users cannot recover the circuit functionality. The resulting FSM can be seen in Figure 2.

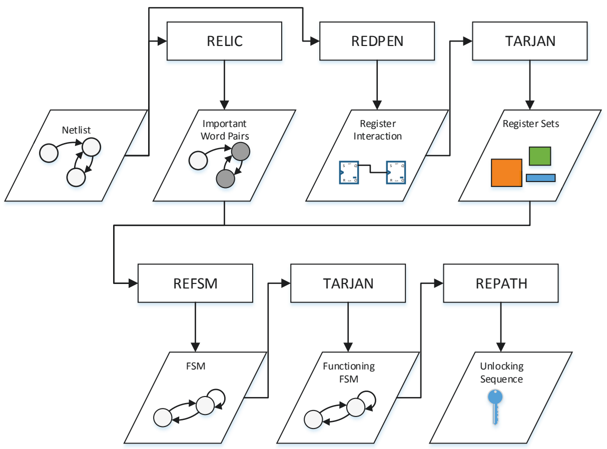

Figure 1: The flow for NETA's logic recovery from a gate-level netlist.

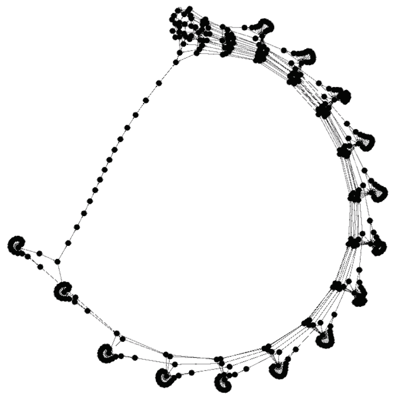

Figure 2: A recovered FSM from a sequentially locked netlist.

At this point it is useful to cover two broad types of FSM encryption that can be employed. The first is that of a statically locked FSM. For such FSM the key to unlock the FSM is unchanging. Some researchers have found methods for breaking this scheme by reducing it to a SAT problem, assuming they have access to an unlocked oracle.

Since the key in a statically locked FSM is unchanging there is a "low" number of possible keys. The likelihood of "accidentilly" unlocking a device brings the topic to the second type of locked FSM. The second type of FSMs are sequentially locked, and ideally after an initial input sequence the FSM will enter a functioning FSM state. Normally each functioning FSM state can reach each other functioning FSM state and no functioning FSM state can reach a non-functioning FSM state. The reachability of a functioning FSM directly implies that the functioning FSM in an FSM should be a "sink" connected component within the SCC graph.

Ideally, if only one sink SCC was found from the FSM, then that SCC is most likely the functioning FSM. However, if there is more than one sink SCC then further analysis is needed to determine the functioning FSM. Another type of sink SCC that could appear would be the black hole states described above. As the probability of entering a black hole state is fairly high, the probabilities of entering a sink SCC provides valuable information on the functionality of that SCC.

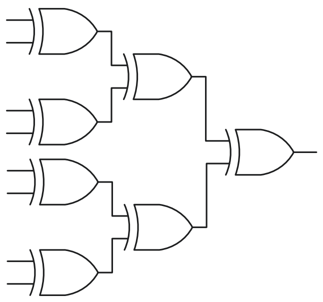

Figure 3: A Simple XOR Tree with 8 inputs

4 REFSM 2.0

Recall that REFSM extracts the logic of a netlist into the more human friendly form of a finite state machine. REFSM had achieved good results in practice and was capable of helping with an array of different problems associated with netlist analysis, such as sequential locking deobfuscation, logic detangling, and hardware Trojan detection. This section will detail a few problems that did not present itself in the original set of netlist benchmarks used to test REFSM and will then introduce an updated version of REFSM.

4.1 XOR Trees

An uncommon structure that could exist in a netlist that causes issues when trying to reveal the potential states reachable within the FSM is a large XOR Tree (see figure 3). The output pin's behavior in a large XOR tree is different from other logic cones in that changing any particular value in the input will flip the output. It means that each input signal has a large effect on the outcome of the subnet. The high input effectiveness over the output is contrary to many logical designs practices whereby only when residing in a particular logical state do certain transitions occur, which is usually implemented using an AND tree or an OR tree. We would like to emphasize the improbability of the XOR Trees being found within logic. The original REFSM, whose function was to analyze the logic of a netlist, overlooked such structures. Due to the XOR Tree's rarity within logic REFSM can perform reasonably well when run on actual circuit designs. Although XOR Trees only exist in specific circuit designs such as encryption circuits, we consider the potential for a netlist to contain XOR Trees as a motivator to one of REFSM's most recent improvements.

The reason why REFSM performance is so stressed by the XOR Tree is due to the method in which REFSM searches for next states. If the output of an XOR tree is found to affect the next state, then REFSM tries to analyze what signals going into the tree force the value to behave in certain manners. Unfortunately the output of an XOR Tree is only decided when all input values are selected. Due to the lack of simple pruning in the XOR Tree case, REFSM will needlessly search the entire space, even if intermediate results have no bearing on the rest of the design. Keeping in mind that an XOR Tree might not have a large effect on the remainder of the netlist. A simple method for checking and removing excess gate structures (including XOR Trees) has been proposed and implemented in REFSM.

4.2 Dominator Trees

The first step we take for the proposed optimization is constructing a Directed Acyclic Graph (DAG) from the original graph. We achieve this by splitting the registers and simultaneously treating them as input and outputs. Certain structures that would not form DAGs include Ring Oscillators (ROs), but they will not be considered as part of the FSM generation, as ROs can add some degree of non-determinism to state transitions. When working with DAGs a sizable amount of graph algorithms become available to computer scientists, which is why this conversion is highly useful.

With the newly created DAG a concept already used with hardware analysis [15] and compiler optimizations [3], the Dominator Tree can very easily be leveraged to help reduce the structure of a given netlist to something more manageable. The following are definitions for the Dominator Trees that will be used in the remainder of our paper:

Definition 1. A Node, u, in a digraph, G, is a **dominator** of Node $v \neq u$, if for all paths in G from the root, r, to v the path contains u.

Definition 2. A Node, u, in digraph, G, is an **immediate dominator** of Node v, if u does not dominate any other dominators of v.

Definition 3. A Node, u, in a digraph, G, is a **Super Dominator Node (SDN)**, if u dominates all nodes in G reachable from u.

Definition 4. A subset, S of nodes of digraph G is **closed** if for all edges, e, in G, we have when e starts in S, then e ends in S. In other words there is no node in S can reach $G - S$.

Other than the split register netlist more preprocessing must be completed prior to using the Dominator Tree. Our optimization's intention of using SDNs is to find large fan-ins that can be simplified to a single signal. That way the SDNs within a netlist can often, but not always, be treated as don't cares when finding all reachable states from an initial state. A minor problem is that SDN dominate the nodes they reach, while reduceable fan-ins that converge at a single input dominate the signals that reach them.

Rather than using the original gate interaction graph when utilizing the Dominator Tree we use the reverse of the graph. That is the edges in the updated graph move from the netlists output to the input. Lastly, since there exists a root in the dominator definitions we create one extra node being the root of our signal graph. This added node is directly connected to all "output" signals in the original graph. It has been shown that extracting a tree that contains the immediate dominator relationship can be done linearly in the number of edges plus nodes for DAGs [5]. For our purposes we compute the Dominator Tree prior to analyzing any states and keep this result throughout the state exploration.

In the example DAG, Figure 4, we have 8 nodes. In the above example the root node is a. For this reason a is the a dominator

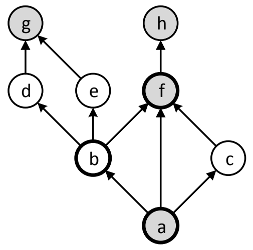

Figure 4: An example of a DAG where the Super Dominant Nodes are the Gray Nodes, and the nodes that dominate some node are bold.

of every other node. The node f has only h in its fan-in, which is definitely dominated by f. Thus f is an SDN. On the other hand the nodes reachable from b are $\{d, e, f, g, h\}$, but f is not dominated by b, because there exists a path to the root a without passing through b. It should be mentioned that b is a dominator of d, e, and g, but b is not an SDN. The other non-SDNs within the example graph is c, d, and e.

Some analysis can be done on the resulting dominator and dominated relationship. By definition all "input" signals which become terminal nodes in the updated DAG are always SDNs. The root is always an SDN, because it dominates everything aside from itself either directly or indirectly. The relationships between nodes and their immediate dominators form a tree, where the inserted root of the DAG is the root of the tree (see Figure 5). The reason for the tree like structure is

(1) The root dominates all nodes. This implies that all nodes have at least one immediate dominator, except the root. Additionally, no node should have more than one immediate domintor, since the graph is acyclic. The immediate dominator can represent a node's parent.
(2) These parents will not form a cycle (even without the DAG restriction the immediate dominators would be acyclic).

This means that each node has an immediate dominator and following a chain of immediate dominators will eventually reach the root. According to the definition, in order to determine if a node u is an SDN, we would need to check if all nodes reachable from u are directly or indirectly dominated. It can be shown that a node u is an SDN if the set of reachable nodes from u is equal to the set of nodes in u's subtree in the Dominator Tree. We propose to use the information about the in and out degree in a subtree of the Dominator Tree to solve the SDN.

4.3 Sub-Dom Trees

First, we need to determine whether the nodes from a subtree in the Dominator Tree is *closed* in the original DAG. When a subgraph is closed the sum of the in degrees and the sum of the out degrees within the subgraph will be equal. A non-closed dominator subtree implies there exists an edge that starts in the subtree of the original DAG and ends outside the DAG. This extra edge will cause the sum of the in degree and out degree sums of the nodes within the subgraph to be different. Note that edges entering the subgraph do not prevent a subgraph from being closed. One "possibility" is

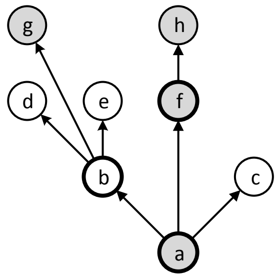

Figure 5: The resulting Dominator Tree extracted from the original example DAG.

Algorithm 1 Use the in (*in*) and out (*out*) degree of each node and the Dominator Tree (*DT*) to find the SDNs

```
function SupderDomNodeFinder($in, out, DT$)
    $SDN \leftarrow \emptyset$
    $n \leftarrow$ NumNodes($DT$)
    $InSum \leftarrow [0] \times n$
    $OutSum \leftarrow [0] \times n$
    $Order \leftarrow$ TopoOrder($DT$)
    for $node \in$ Reverse $Order$ do
        $InSum[node] \leftarrow InSum[node] + in[node]$
        $OutSum[node] \leftarrow OutSum[node] + out[node]$

        if $InSum[node] - in[node] = OutSum[node]$ then
            $SDN \leftarrow SDN \cup \{node\}$
        end if

        $parent \leftarrow DT'$s parent of $node$
        $InSum[parent] \leftarrow InSum[parent] + InSum[node]$
        $OutSum[parent] \leftarrow OutSum[parent] + OutSum[node]$
    end for
    return $SDN$
end function
```

that an edge could increase the in degree of a subgraph, but not the out degree, which would make a node appear to be an SDN, when in reality the subgraph is not closed. This "extra" edge is actually impossible except when incident to the subroot of the sub-tree, in which case it can easily, and in constant time be removed.

Thus the original problem can be reduced to checking if the in degree sum of a subtree is equal to its out degree sum, where the only node whose in degree will not be contributed to the sum is the subroot. To quickly compute this value we will utilize dynamic programming. We will not double count degrees, because we have a tree structure, and there is only one path from a subroot to any reachable node. Algorithm 1 presents a high level implementation.

For the original sample graph you can see the in degree and out degree for each node in Figure 6. The computed sum of the subtree's in degree and out degree for the sample graph is in Figure 7. The actual in degree and out degree sum, with the root's in degree removed, for the sample graph can be seen in Figure 8. Note that the nodes where the sums are equal were the SDNs in the original graph. This adjusted sum comparison can be seen in line 11 of Algorithm 1

4.4 Experimental Results

To test the capabilities of REFSM 2 when compared to the original REFSM we used a few realistic designs, but again due to the rarity

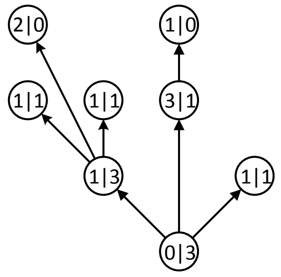

Figure 6: The in degree and out degree for individual node.

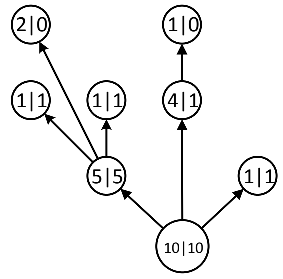

Figure 7: The in degree and out degree sum for the subtree (including the in degree for the subroots).

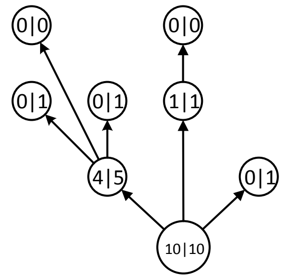

Figure 8: The in degree and out degree sum for the actual subtrees.

of an XOR tree affecting the logic of a netlist, no significant changes were detected on the few practical designs as seen in Table 1. The second attempt to distinguish REFSM vs REFSM 2 involved making some proof of concept netlists that contained XOR trees ranging in sizes from 2 to 64 inputs. In theory REFSM would not be able to work on XOR trees quickly with more than about 30 inputs. The results in Figure 9 show that the original REFSM had even more trouble than what had been estimated. It should be noted that any run that took more than an hour was stopped. The worse than expected performance of REFSM is probably a result of its significant constant factor in its runtime. The results were collected on a machine with 132 GB of memory and an Intel(R) Core(TM) i7-4770 processor.

Testing Circuits	Total Gates	Total States	Total Transitions	REFSM Time (s)	REFSM 2 Time (s)
Locked AES	16766	386	752	14.804	14.436
S1196	331	525	127025	32.362	31.778
UART	168	390	195	0.242	0.244
MC8051	6590	5	13	114.09	113.92
MSP430 UART FSM	4481	6	22	9.59	10.09
MSP430 Mem FSM	4481	4	8	0.20	0.23

Table 1: The results of REFSM and REFSM 2 on a few designs from various sources.

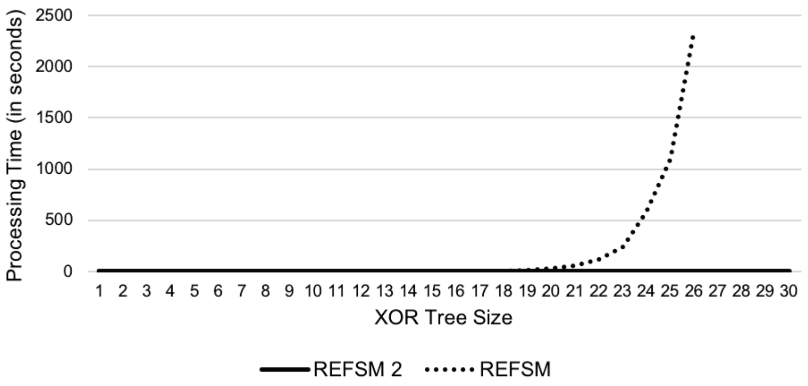

Figure 9: The results of the REFSM vs REFSM 2 when run on XOR tree based FSMs of varying input sizes.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced our NETA toolset and its ability to extract rare occurring states through netlist decryption. Additionally, we discussed linear time precomputations that have the potential to improve our previous methods. Experimental results were collected to demonstrate the potential changes in runtime. In our future work, we plan to further extend REFSM as well as all other tools in the NETA suite by finding and implementing more optimizations. Considering the fact that IP consumers are vulnerable to attacks from within ICs, we will also focus on providing more netlist reverse engineering tools for assessing their technology.

ACKNOWLEDGEMENT

The work is partially supported by the National Science Foundation (NSF-1812071).

REFERENCES

[1] Jacob Couch, Elizabeth Reilly, Morgan Schuyler, and Bradley Barrett. 2016. Functional block identification in circuit design recovery. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 75–78.
[2] Yu-Yun Dai and Robert K Braytont. 2017. Circuit recognition with deep learning. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*. IEEE, 162–162.
[3] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[4] Matthew Hicks, Murph Finnicum, Samuel T King, Milo MK Martin, and Jonathan M Smith. 2010. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 159–172.
[5] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141.
[6] Wenchao Li, Zach Wasson, and Sanjit A Seshia. 2012. Reverse engineering circuits using behavioral pattern mining. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 83–88.
[7] Travis Meade, Yier Jin, Mark Tehranipoor, and Shaojie Zhang. 2016. Gate-Level Netlist Reverse Engineering for Trojan Detection and Hardware Security. In *The IEEE International Symposium on Circuits and Systems (ISCAS)*. 1334–1337.
[8] Travis Meade, Kaveh Shamsi, Thao Le, Jia Di, Shaojie Zhang, and Yier Jin. 2018. The Old Frontier of Reverse Engineering: Netlist Partitioning. *Journal of Hardware and Systems Security* 2, 3 (2018), 201–213.
[9] Travis Meade, Shaojie Zhang, and Yier Jin. 2016. Netlist reverse engineering for high-level functionality reconstruction. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, 655–660.
[10] Travis Meade, Shaojie Zhang, and Yier Jin. 2017. IP protection through gate-level netlist security enhancement. *Integration, the VLSI Journal* 58 (2017), 563–570.
[11] Travis Meade, Shaojie Zhang, and Yier Jin. 2018. NETA: Netlist Analysis Toolset. (2018). https://github.com/jinyier/NetA.
[12] Travis Meade, Zheng Zhao, Shaojie Zhang, David Pan, and Yier Jin. 2017. Revisit sequential logic obfuscation: Attacks and defenses. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 1–4.
[13] Jordan Robertson and Michael Riley. 2018. The big hack: how china used a tiny chip to infiltrate U.S. companies. *Bloomberg Businessweek* (4 Oct 2018).
[14] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. 2010. A highly efficient method for extracting FSMs from flattened gate-level netlist. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2610–2613.
[15] Alex Shye, Matthew Iyer, Vijay Janapa Reddi, and Daniel A Connors. 2005. Code coverage testing using hardware performance monitoring support. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 159–163.
[16] Pramod Subramanyan, Nestan Tsiskaridze, Wenchao Li, Adria Gascón, Wei Yang Tan, Ashish Tiwari, Natarajan Shankar, Sanjit A Seshia, and Sharad Malik. 2014. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Trans. Emerging Topics Comput.* 2, 1 (2014), 63–80.
[17] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.
[18] Edward Tashjian and Azadeh Davoodi. 2015. On using control signals for word-level identification in a gate-level netlist. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 78.
[19] Michael Werner, Bernhard Lippmann, Johanna Baehr, and Helmut Gräb. 2018. Reverse engineering of cryptographic cores by structural interpretation through graph analysis. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. IEEE, 13–18.