Trojan Localization using Symbolic Algebra

Farimah Farahmandi, Yuanwen Huang and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, USA

***Abstract*—Growing reliance on reusable hardware Intellectual Property (IP) blocks, severely affects the security and trustworthiness of System-on-Chips (SoCs) since untrusted third-party vendors may deliberately insert malicious components to incorporate undesired functionality. Malicious implants may also work as hidden backdoor and leak protected information. In this paper, we propose an automated approach to identify untrustworthy IPs and localize malicious functional modifications (if any). The technique is based on extracting polynomials from gate-level implementation of the untrustworthy IP and comparing them with specification polynomials. The proposed approach is applicable when the specification is available. Our approach is scalable due to manipulation of polynomials instead of BDD-based analysis used in traditional equivalence checking techniques. Experimental results using Trust-HUB benchmarks demonstrate that our approach improves both localization and test generation efficiency by several orders of magnitude compared to the state-of-the-art Trojan detection techniques.**

I. Introduction

Intellectual Property (IP) outsourcing is a widely used practice in System-on-Chip (SoC) design methodology to reduce the time to market and overall cost. However, it raises major security risks as the attacker can embed malicious components in third-party IPs. Such malicious components, widely known as hardware Trojans, may affect the correct behavior and defeat the trustworthiness of the design by leaking protected information such as secret keys. Hardware Trojans consist of two parts: a trigger and a payload. The trigger is a set of conditions such that their activation deviates the desired functionality from the specification and their effects are propagated through the payload. The adversary designs trigger conditions such that they are satisfied in very rare situations and usually after long hours of operation [1]. Conventional structural and functional testing methods are not effective to activate trigger conditions since there are many possible Trojans and it is not feasible to construct a fault models for each of them. As a result, existing EDA tools are incapable of detecting hardware Trojans and differentiating between trustworthy third-party IPs and untrustworthy ones.

There has been a lot of research on hardware Trojan detection using logic testing and side channel analysis [1], [2], [3]. Logic testing focuses on generating efficient tests to activate a Trojan and check the primary output values of specification and circuit-under-test to detect Trojan. Side channel analysis focuses on the difference of the side-channel signature between the golden circuit and Trojan infected circuit. These two types of methods answer the question of whether a circuit is infected with Trojan, but they cannot identify the location of the Trojan. Approaches based on structural/functional analysis [4], [5], [6], [7] have been proposed to identify/localize the malicious logic. Unused Circuit Identification (UCI) [4] finds for unused portions in the circuit and flag them as malicious. Sturton et al. show that many other types of malicious circuits can evade the detection of the UCI algorithm [8]. The FANCI approach [5] was proposed to flag suspicious nodes based on the concept of control values. However, FANCI flags about 1-8% of all nodes, which might be too many suspicious candidates for experts to analyze for a large circuit. Moreover, FANCI returns a set of suspicious nodes even when the circuit is Trojan free. A recent work by Oya et al. [6] manually crafted templates for Trojans and was successful in using these templates to identify Trojans in TrustHUB benchmarks [9]. Unfortunately, this approach is applicable only for specific types of Trojans, therefore, it is not suitable to detect other types of Trojans that are not covered by their templates.

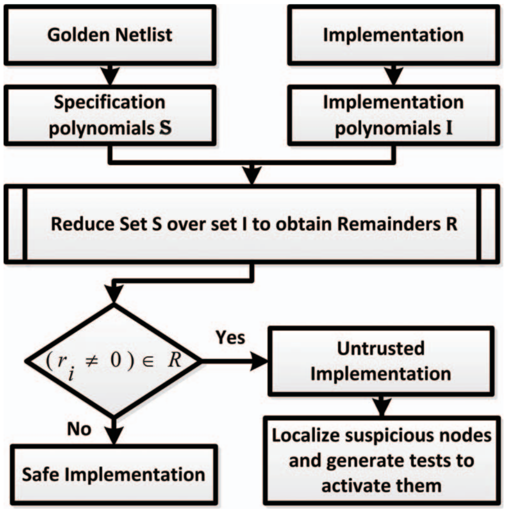
Golden Netlist
Implementation
Specification polynomials S
Implementation polynomials I
Reduce Set S over set I to obtain Remainders R
$(r_i \neq 0) \in R$
Yes
No
Untrusted Implementation
Safe Implementation
Localize suspicious nodes and generate tests to activate them

Fig. 1: The proposed hardware Trojan localization flow

In order to address the above mentioned challenges, we propose a design-time formal method to localize and activate Trojans between two versions of a design. Suppose that we have a golden model of a design (specification), and a modified version (implementation) of it (after performing some non-functional changes such as doing synthesis, adding clock trees, scan chain insertion etc.). We would like to make sure that there is no hardware Trojan inserted to the design during the non-functional changes. In other words, our goal is to make sure that two versions of a design are functionally equivalent (nothing more, nothing less) and an adversary cannot hide hard-to-detect malicious modifications during design transformations. Our approach is scalable since it uses polynomial based manipulation instead of Binary Decision Diagrams (BDD [10]).

We propose a formal method based on symbolic algebra to detect potentially malicious modifications in the implementation. Our method is based on extraction of functional polynomial [11] from gate-level IPs. Figure 1 presents the overview of our proposed methodology. We extract a set of polynomials from the specification ($\mathbb{S}$). We also derive a set of polynomials ($\mathbb{I}$) from the implementation. Finally, we check the equivalency between two sets $\mathbb{S}$ and $\mathbb{I}$ based on Gröbner Basis Reduction. Each of the polynomials from the specification f_{spec_i} is reduced over a set of corresponding polynomials $\mathbb{I}$ and a set of remainders $\mathbb{R}$ is generated. From

978-1-5090-1558-0/17/$31.00 ©2017 IEEE

symbolic computer algebra, it is known that when $r_i = 0$, gates in Rg ((set of gates that contribute in reduction of polynomial f_{spec_i} is called region Rg)) have successfully implemented f_{spec_i} and it guarantees that all gates in Rg are safe [12]. Any $(r_i \neq 0) \in \mathbb{R}$ shows a suspicious functionality in the corresponding region Rg and all of the gates in Rg are suspicious candidates. The malicious nodes can be pruned by removing the safe gates from the suspicious candidates. When all of r_i's are equal to zero, the implementation is Trojan free. The proposed method can recognize the Trojan-free implementation from the Trojan-inserted one. Our method reports a few gates to indicate the presence of a malicious activity (change of functionality) in the implementation. Since the number of malicious gates is very small, our approach is amenable for an exhaustive test generation to activate the Trojan. Our method is applied on Trust-HUB benchmarks [9] and the experimental results show the effectiveness of our approach compared to existing methods.

The remainder of the paper is organized as follows. We discuss related work in Section II. Section III gives an overview of equivalence checking using polynomials. Section IV discusses our framework for hardware Trojan localization and detection. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. Related Work

To detect a Trojan with logic testing, the test should not only be able to satisfy the trigger condition to activate the Trojan, but it should also be able to propagate the Trojan payload to primary (observable) outputs. Based on the fact that the trigger condition usually has extremely low probability, the traditional Automatic Test Pattern Generation (ATPG) based method for functional testing cannot fulfill the task of Trojan activation and detection. Bhunia et al. [1] proposed the multiple excitation of rare occurrence (MERO) approach to generate more effective tests to increase the probability to trigger the Trojan. Recent work by Saha et al. [13] improved MERO to get higher detection coverage by identifying possible payload nodes. They used genetic algorithm to assist ATPG to generate high quality tests which can propagate the possible payload values to primary outputs. However, test generation does not have the capability of localizing a Trojan to find malicious gates. Side channel analysis focuses on the side channel signatures of the circuit [2], [3], which avoids the limitations (low trigger probability and propagation of payload) of logic testing. However, the abnormality in side channel signatures for Trojan circuit is sensitive to measurement noise and process variation, which makes side channel analysis not effective on large circuits. Narasimhan et al. [15] proposed the temporal self-referencing approach on large sequential circuits. Recently, Yuanwen et al. [14] proposed the multiple excitation of rare switching (MERS) approach to combine the advantages of logic testing and side channel analysis.

Waksman et al. [5] proposed the FANCI approach to identify suspicious nodes using Boolean functional analysis. Their approach will report about 1-8% of all nodes as suspicious candidates, which still needs to be analyzed by experts to further localize the Trojan. Moreover, this method is likely to report a set of suspicious candidates for even trusted IP blocks. In other words, this method cannot uniquely identify whether a hardware Trojan is inserted. A score-based approach [6] was recently proposed by Oya et al., which can identify the Trojans in Trust-HUB benchmarks by using pre-defined Trojan templates. However, an adversary can design so many different Trojan circuits and this approach is likely to fail while facing new Trojan designs. Banga et. al. [16] proposed a four-step technique to localize suspicious components in third-party IPs. However, this approach fails for large circuits due to the complexity limitations of SAT solvers. Rajendran et al. [18] proposed a formal method to detect unauthorized corruption of critical data in the design such as secret keys. However, the presented method requires to have the information about valid ways to access critical data which is not trivial. Moreover, it considers only one form of Trojan in a design, and cannot consider a wide variety of Trojans that are possible in SoC IPs. We propose a fully automated and efficient Trojan localization method to address the above mentioned challenges.

III. Background: Equivalence Checking using Gröbner Basis Reduction

The equivalence checking problem can be efficiently mapped to ideal membership testing for arithmetic circuits [19], [20], [12]. To be able to apply Gröbner basis theory, specification is modeled as polynomial f_{spec} and implementation is converted to a set of polynomials $\mathbb{I} = \{f_1, f_2, ..., f_s\}$ and ideal I is constructed as $I =< \mathbb{I} >=< f_1, f_2, ..., f_s >$. Set $\mathbb{I}$ is derived in a way that every pair (f_i, f_j) has relatively prime leading monomials. Thus, set $\mathbb{I}$ is also Gröbner basis (G) of ideal I ($\mathbb{I} = G$). To check equivalence between specification and implementation, f_{spec} is reduced over set G. If the remainder is zero, it shows that the implementation has correctly implemented the specification. In other words, the specification and implementation are equivalent.

The functionality of logic gates (such as AND, OR, XOR, NOT and buffer) can be represented by polynomials such that the inputs and output signals of gates act as variables of the corresponding polynomial. Each variable x_i which appears in a circuit polynomial, belongs to $\mathbb{Z}_2$ where $({x_i}^2 = x_i)$. The basic logic gates are described using polynomials shown in Equation 1. Polynomial of complex gates can be derived by combining the equations of basic gates.

$$\begin{aligned}
&n_1 = NOT(x_1) \rightarrow n_1 = (1 - x_1),\\
&n_2 = BUFF(x_1) \rightarrow n_2 = x_2,\\
&n_3 = AND(x_1, x_2) \rightarrow n_3 = x_1.x_2,\\
&n_4 = OR(x_1, x_2) \rightarrow n_4 = x_1 + x_2 - x_1.x_2,\\
&n_5 = XOR(x_1, x_2) \rightarrow n_5 = x_1 + x_2 - 2.x_1.x_2
\end{aligned} \tag{1}$$

Specification of an arithmetic circuit (like multipliers, adders and their combinations) can be represented as one polynomial f_{spec}. Polynomial f_{spec} represents the word-level abstraction of arithmetic circuits functionality using primary inputs and primary outputs as variables. For example, the specification of a n-bit adder with primary inputs $A = \{a_0, a_1, ..., a_{n-1}\}$ and $B = \{b_0, b_1, ..., b_{n-1}\}$ and primary output $Z = \{z_0, z_1, ...z_n\}$ can be formulated as $(2^n.z_n + ... + 2.z_1 + z_0) - ((2^{n-1}.a_{n-1} + ... + 2.a_1 + a_0) + (2^{n-1}.b_{n-1} + ... + 2.b_1 + b_0)) = 0$ where $\{a_i, b_i, z_i\} \subset \{0, 1\}$.

Suppose that we want to make sure an arithmetic circuit implements correctly its specification. In other words, we want to verify that there are no functional errors in the arithmetic circuit. To test it, the specification polynomial f_{spec} is reduced over implementation polynomials $\mathbb{I}$ by considering an order. Then, the result of the reduction is considered. The non-zero result demonstrates the possibility of a threat. Example 1 shows the procedure to verify functionality of a full-adder.

Example 1: Suppose that we want to verify the functional correctness of a full-adder implementation shown in Figure 2. The specification can be formulated as: $(2.C_{out}+S-(A+B+C_{in}))$ and each gate in the implementation can be modeled as a polynomial based on Equation 1. The topological order of the circuit (since the circuit is acyclic) is chosen for reduction as $C_{out} > \{S, n_3\} > \{n_2, n_1\} > \{A, B, C_{in}\}$. The reduction starts from the most significant primary output and ends at primary inputs. Variables in the curvy brackets have the same order and they can be reduced in one iteration. Equation 2 shows the reduction process. It can be seen that the final result (remainder) is a non-zero polynomial and we can conclude that the implementation is incorrect. If we use method presented in [21] and change the NAND gate with an AND gate (correct the bug) and redo the procedure, it will lead to a zero remainder.

$$\begin{aligned}
&step_0 : 2.C_{out} + S - A - B - C_{in}\\
&step_1 : S - 2.n_3.n_2 + 2.n_3 + 2.n_2 - A - B - C_{in}\\
&step_2 : 2.n_2.n_1.C_{in} - 4.n_1.C_{in} + n_1 - A - B + 2\\
&step_3(remiander) : 8.A.B.C_{in} - 4.A.C_{in} - 4.B.C_{in} - 2.A.B + 2
\end{aligned} \tag{2}$$

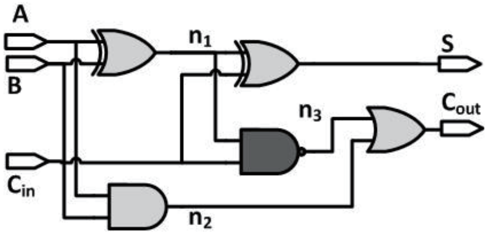

Fig. 2: Faulty gate-level netlist of a full-adder

IV. Trojan Detection and Localization

In order to trust an IP block, we have to make sure that the IP is performing exactly the expected functionality. The approach presented in Section III can be extended to find whether a hardware Trojan, which changes the functionality, has been inserted in a combinational arithmetic circuit. However, applying the same approach on general IPs is limited due to several reasons. First, it is possible that the specification of a general circuit cannot be described as one simple polynomial. Second, the circuit may not be acyclic and loops may exist due to their sequential nature. Third, unrolling may increase the complexity of the problem so the reduction of f_{spec} over implementation polynomials will face polynomial terms explosion. Finally, the Trojan activation may require extremely large number of unrolling steps which may be practically infeasible and also there is no specific information on after how many cycles Trojan will be activated. In order to address these challenges, we present a method to generate polynomials in an efficient way and use them in our proposed algorithm to localize and detect Trojans in third-party IPs. To the best of our knowledge, our proposed approach is the first attempt in utilizing scalable equivalence checking using polynomial manipulation for localization of hardware Trojans. The reminder of this section describes the three important tasks in our framework: polynomial generation, Trojan localization, and test generation for Trojan detection.

A. Polynomial Generation

Suppose that we have two versions of a design, one is a verified IP (specification) and the other is an untrusted third-party IP (implementation) after performing non-functional transformations. Our goal is to detect whether an adversary has inserted hard-to-detect hardware Trojan during non-functional changes and has made undesired functional changes. For example, a design house may send their RTL design for synthesis or adding low-power features to a third party vendor. Once the third-party IP comes back (after synthesis or other functionality-preserving transformations), it is crucial to ensure the trustworthiness of these IPs.

In the method presented in Section III, specification is modeled as one polynomial; however, here we generate a set of polynomials $\mathbb{S}$ representing the functionality of the golden IP to be able to apply Gröbner basis theory for hardware Trojan localization problem. The specification is partitioned into several regions and each region is converted to a polynomial. The output of each region is either inputs of a flip-flop (clock, enable, reset and etc.) or one of the primary outputs. The inputs of a region are either from primary inputs or inputs/outputs of flip-flops. In other words, we generate polynomials for regions which are limited to flip-flops' boundaries. Then, corresponding equations (based on Equation 1) of gates inside a region are combined together to construct one polynomial representing the functionality of the region.

Algorithm 1 Polynomial generation algorithm

```
1: procedure POLYNOMIAL-GENERATION
2:     Input: Circuit Graph $Gr$, $L_{out}$ and $L_{in}$
3:     Output: Polynomials $\mathbb{S}$
4:     Region = {}
5:     for each gate $g_i \in Gr$ where its output ∊ $L_{out}$ do
6:         Region.add($g_i$)
7:         for all inputs $g_j$ of $g_i$ do
8:             if !($g_j$ ∊ $L_{in}$) then
9:                 Region.add($g_j$)
10:                Call recursively for inputs of $g_j$ over $Gr$
11:        $f_i$ = convertToPolynomial(Region)
12:        $\mathbb{S} = \mathbb{S} \cup f_i$
13:        Region = {}
14: return $\mathbb{S}$
```

Algorithm 1 shows how we extract set $\mathbb{S}$. The specification is converted to a graph where each vertex is a gate (g_i). The algorithm takes the circuit graph Gr, list (L_{out}) of allowed output variables (flip-flops' inputs and primary outputs) and list (L_{in}) of allowed input variables of a region as inputs and returns a set of polynomials $\mathbb{S}$ as its output. The algorithm chooses a gate for which output belongs to L_{out} and goes backward recursively until it reaches the gate g_j, whose input comes from one of the variables from L_{in} (line 5-10). The algorithm marks all the visited gates as a "*Region*". The selected region may contain all of the basic gates except flip-flops. Then, the *Region* is converted to a polynomial f_i by combining corresponding polynomials of the gates residing in *Region*, f_i is added to set $\mathbb{S}$ (line 11-12).

Example 2: Suppose that the circuit shown in Figure 3 is a part of a verified IP block and we want to use it as our specification. Algorithm 1 is applied on it and the polynomials are shown as: $\mathbb{S} = \{f_{spec_1} : n_1 - (-2.A.n_2 + n_2 + A), f_{spec_2} : Z - (1 - n_1.B)\}$. Since the circuit shown in Figure 3 contains one primary output and one flip-flop, the Algorithm 1 extracts two specification polynomials for this circuit.

Similarly, the implementation polynomials $\mathbb{I}$ are driven by modeling every gate except flip-flops from the untrusted design as a polynomial based on Equation 1 and Algorithm 1.

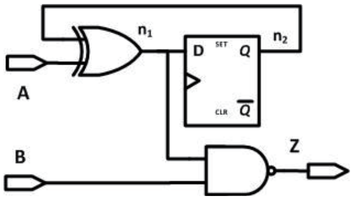

Fig. 3: A part of a sequential circuit

In order to reduce the number of generated implementation polynomials, we partition implementation to fanout-free cones (set of gates that are directly connected together) and convert each fanout-free region as one polynomial. In other words, $\mathbb{I}$ contains a set of polynomials where each polynomial represents a fanout-free cone.

Example 3: The circuit shown in Figure 4 is the Trojan-inserted implantation of the specification shown in Figure 3 (gate 6 is the Trojan trigger and gate 7 is the payload). Gates in same pattern belong to a common fanout-free cone. As a result, set $\mathbb{I}$ is computed by Algorithm 1. Each polynomials is corresponding to one fanout-free cone.

$$\begin{aligned}\mathbb{I} = \{&n_1 - (n_2.w_4.A - n_2.w_4 + w_4 - n_2.A + n_2),\\ &w_4 - (A - n_2.A),\\ &Z - (n_1.w_4.C.B - n_1.w_4.C - n_1.B + 1)\}\end{aligned} \tag{3}$$

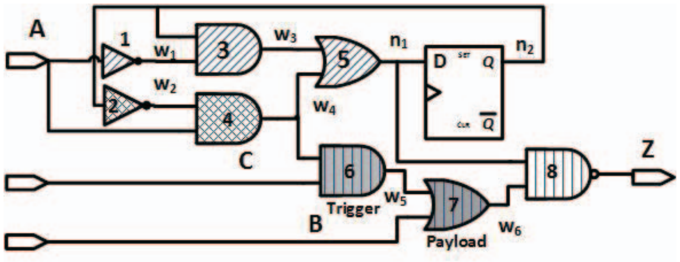

Fig. 4: A Trojan-inserted implementation of circuit in Figure 3

B. Trojan Localization

We generate the set $\mathbb{S}$ and $\mathbb{I}$ as described in Section IV-A. We assume that the name of flip-flops, primary inputs and primary outputs are the same between implementation and specification or the name mapping can be done. We also assume that no re-timing has been performed. These are valid assumptions in many scenarios involving third-party IPs. The equivalence of two sets $\mathbb{S}$ and $\mathbb{I}$ is checked to find any suspicious functionality which may serve as a Trojan.

To detect a Trojan, we need to reduce each polynomial f_{spec_i} from set $\mathbb{S}$ over a subset of polynomials from set $\mathbb{I}$ to check membership of every polynomial f_{spec_i} in Ideal I constructed from polynomials from set $\mathbb{I}$ ($I =< \mathbb{I} >$). To perform that, all of the polynomials from $\mathbb{I}$ are hashed based on their leading terms (which contains a single variable and this variable represents the output of the corresponding gate). Every variable from $f_{spec_i} \in \mathbb{S}$ is replaced with the corresponding functionality of that variable from $\mathbb{I}$ polynomials. The process continues until f_{spec_i} is reduced either to zero polynomial or a remainder polynomial which contains primary inputs as well as flip-flop's inputs/outputs. The non-zero remainder indicates that implementation does not correctly implement the functionality of f_{spec_i} and that part of the implementation is suspicious. Note that, based on Gröbner basis theory, when the remainder is zero for a specific region, we can be certain that the region is safe. In other words, it is not possible for a smart attacker to insert malicious gates in a way that the remainder becomes zero.

Example 4: Consider we want to measure the trust in the circuit shown in Figure 4, which is the untrustworthy implementation of design shown in Figure 3. Specification Polynomials shown in Example 2 are reduced over implementation polynomials as shown in Equation 3. The result of the reduction is stored in set $\mathbb{R}$. Each f_{spec_i} produces one remainder r_i that can be either zero or a non-zero polynomial. Gates $\{1, 2, 3, 4, 5\}$ implement functionality of an XOR gate (these gates are equivalent to XOR gate shown in Figure 3). Thus, the remainder r_1 is zero and it means that the region containing gates $\{1, 2, 3, 4, 5\}$ implements the f_{spec_1} correctly. However, the non-zero remainder r_2 presents the fact that there are malicious components in implementation of f_{spec_2} and the region containing gates $\{2, 4, 6, 7, 8\}$ is suspicious.

$$\begin{aligned}&f_{spec_1} : n_1 + 2.A.n_2 - n_2 - A\\ &step_{11} : n_2.w_4.A - n_2.w_4 + w_4 + n_2.A - A\\ &step_{12}(r_1) : 0\\ &f_{spec_2} : Z + n_1.B - 1\\ &step_{21} : n_1.w_4.C.B - n_1.w_4.C\\ &step_{22}(r_2) : -1.n_1.A.C + n_1.n_2.A.C + A.B.C.n_1 - A.B.C.n_1.n_2\end{aligned} \tag{4}$$

By using the proposed approach, a set of malicious regions are identified. Suppose the adversary inserts some extra flip-flops as part of Trojans. These buggy flip-flops does not have any correspondence in the specification. In other words, there is no f_{spec_i} which describes their inputs' functionality. Therefore, the corresponding region in the implementation is also considered as a suspicious region. However, scan-chain flip-flops can easily be detected and removed from suspicious candidates because of their structures.

The proposed method formally identifies the regions (between flip-flops boundaries) of the implementation that are safe and the regions that have suspicious functionality. The adversary usually insert the Trojan in deep levels of the circuit. Therefore, the regions that actually contain the Trojan can be very large and may include many gates (order of hundreds or thousands of gates). In order to improve our approach further, we propose an algorithm to identify the gates that most likely are responsible for the malicious activity. Since we know which regions are Trojan-free (based on remainder as zero), we remove the gates which are contributing in construction of these regions from suspicious regions. In other words, we have formally proved that some of the regions are trustworthy so the gates that construct these regions are essential for the correct functionality. The safe gates may be inputs of Trigger or payload gates. However, they do not belong to the set of malicious gates. Using this approach, we are able to prune the suspicious regions to contain very small number of gates. This approach guarantees that all of the Trojan trigger and payload's gates are inside the suspicious region. Algorithm 2 shows the proposed procedure.

The algorithm takes the gate-level implementation graph G_r as well as specification and implementation polynomials as inputs, and in case the implementation contains malicious components, it returns a set of suspicious gates as output. The algorithm takes each of specification polynomials and reduces them one by one over corresponding polynomials from set $\mathbb{I}$. Each f_{spec_i} may be reduced using several gates g_j and the result of the reduction is stored in r_i (line 4-5). The used gates are marked to keep track of the gates that

Algorithm 2 Hardware Trojan localization algorithm

```
1: procedure TROJAN–LOCALIZATION
2:     Input: Circuit implementation $G_r$, $\mathbb{I}$ and $\mathbb{S}$
3:     Output: Suspicious gates $G_t$
4:     for each $f_{spec_i} \in \mathbb{S}$ do
5:         $r_i$ = reduction of $f_{spec_i}$ over $f_js \in \mathbb{I}$
6:         $R_i = R_i \cup all\ g_js\ where\ f_j = func(g_j)$
7:         mark all $g_i$s as used
8:         if $(r_i! = 0)$ then
9:             $R_{TrjIn} = R_{TrjIn} \cup R_i$
10:        else
11:            $R_{TrjFree} = R_{TrjFree} \cup R_i$
12:    for each gate $g \in R_{Trjfree}$ do
13:        remove g from $R_{TrjIn}$
14: return $G_t$ = remaining in $R_{TrjIn} \cup unused\ gates$
```

are utilized to implement the circuit (line 6). If r_i is equal to zero, it means that all of the g_is are safe and they are stored as safe gates ($R_{TrjFree}$), otherwise, all g_is are stored as suspicious candidates (line 7-11). Every $r_i = 0$ shows that all of the gates used in construction of functionality of the corresponding f_{spec_i} are safe. Therefore, to narrow down the potential suspicious gates, the gates of G_r which appeared in $R_{TrjFree}$ are removed from R_{TrjIn} (line 12-13). Note that, gates in both of $R_{TrjFree}$ and R_{TrjIn} belong to the implementation G_r. All of unused gates should also be considered as malicious candidates, so the union of the remaining gates in the R_{TrjIn} and unused gates are returned as likely malicious gates (G_t). If all of the r_is are zero, the implementation is safe and there is no Trojan inside the implementation.

Algorithm 2 identifies the trust level of a third-party IP and in case of existence of hardware Trojan, it returns a very small number of gates as suspicious candidates. This algorithm guarantees that all of the actual Trojan trigger and payload gates are inside the set G_t.

Example 5: Applying Algorithm 2 on the circuit shown in Figure 4 will result in non-zero remainder for region containing gates $\{2, 4, 6, 7, 8\}$. However, the zero remainder of f_{spec_1} shows that gates $\{1, 2, 3, 4, 5\}$ are safe and they are vital to construct the functionality of signal n_1. Therefore, we remove gates $\{2, 4\}$ from potential candidates and gates $\{6, 7, 8\}$ remain as suspicious.

C. Trojan Activation

As shown in Example 5, the small suspicious region still contains some safe gates which are dedicated to the correct functionality in the absence of the Trojan (in Example 5, gate 8 is benign but it is reported as suspicious node). In other words, these safe gates are only used to construct the functionality of one specific primary output or Flip-Flop's input. Thus, they won't be removed in the process of pruning safe gates from suspicious regions since they are not contributing in functionality of other primary outputs or flip-flop's inputs. To be able to detect the exact gates which are responsible for trigger and payload parts of Trojan, we generate tests to activate the Trojan. Since the number of suspicious gates are small enough, we try to activate each node in the suspicious gates and check whether the generated test activates the Trojan. We use an ATPG to generate the directed tests. If none of the tests detects the Trojan, we generate test to activate two of the nodes at the same time. We continue the process until one of the tests activates the Trojan. This approach is feasible due to the fact that the number of suspicious nodes that are reported using our proposed approach is very small.

Example 6: We are trying to activate the Trojan shown in Figure 4. From Example 5, we know that gates $\{6, 7, 8\}$ are suspicious. As shown in Figure 4, Trojan will be triggered when output of gate 6 (w_5) becomes true and B is zero at the same time. In other words, gate 8 of the implementation receives one as its second input (w_6) while in the specification, the second input of the NAND gate receives zero. These conditions cause difference between specification and implementation. To propagate the effect of Trojan's condition activation, n_1 should be one since $n_1 = 0$ makes output $Z = 1$ independent of second input's value and it will mask the Trojan effect. The test vectors that activate Trojan are as follows (we assume the initial value of n_2 is equal to 0): $A = 1, B = 0, C = 1$.

V. EXPERIMENTS

A. Experimental Setup

The Trojan localization algorithm was implemented in a Java program and experiments were conducted on PC with Intel Processor E5-1620 v3 and 16 GB memory. We have tested our approach using widely used trust-HUB benchmarks [9] consisting of combinational and sequential Trojan triggers and payloads that change the functionality of the design. The Trojan-Free designs are considered as specification. To show that our methodology is orthogonal to design structures and library format, we synthesized Trojan-inserted benchmarks with Xilinx synthesis tool and used them as implementation (we just map flip-flops' inputs/output names). Specification is partitioned into several regions and each region is represented using one polynomial. These polynomials can be reduced over implementation polynomials independently. Therefore, we used a parallel version of Algorithm 2 to implement our method. We also used logic reduction based rewriting schemes presented in [23] to improve the equivalence checking time. We compared our results with most relevant Trojan localization work [5]. Since our approach essentially performs equivalence checking, we also compared with an equivalence checking tool *"Formality"* [22] which has been designed to check the equivalence between two versions of a design to demonstrate the efficiency of our work. Formality is an commercial tool that tries to detect potential functional changes between two versions of a design when the designers making non-functional changes.

Formality compares the points between two designs and tries to match them using different algorithms including name-based matching and non-name based matching algorithms. Based on formality's user guide [24], it first compares the points based on their exact names. Then, it tries to perform case-insensitive name mapping or filtering out some characters. Name matching can also be done through mapping driven/driving nets (name of nets) of points. In the second phase, it attempts to match the remaining unmatched points using topological analysis of the unmatched cones. In other words, it matches two points with different names if they have equivalent structures. The final step is signature analysis which is based on generating functional and topological signatures. Functional signatures use random patterns simulation to generate primary outputs' data or register's output data to

TABLE I: Trojan Localization using Trust-HUB benchmarks.

Benchmark			FANCI [5]	Formality [22]	Our Approach				False Positive%			Improvement	
Type	#Gates	#TrojanGates	#SuspGates	#SuspGates	#SuspGates	#Spolys	#Ipolys	CPU time(s)	our	[5]	[22]	[5]	[22]
RS232-T1000	311	13	37	214	13	62	186	0.67	0	24	201	*	*
RS232-T1100	310	12	36	213	14	61	189	0.86	2	24	201	12x	100.5x
S15850-T100	2456	27	76	710	27	592	1888	1.13	0	49	683	*	*
S38417-T200	5823	15	73	2653	26	1667	5004	3.12	11	58	2638	5.27x	239.8x
S35932-T200	5445	16	70	138	22	1778	4441	3.18	6	54	122	9x	20.33x
S38584-T200	7580	9	85	47	11	840	3905	4.74	2	76	38	38x	19x
Vga-lcd-T100	70162	5	706	**	22	2426	7572	38.97	17	701	**	41.23x	**

"*" indicates our approach does not produce any false positive gates (infinite improvement).
"**" shows the cases that *Formality* could not detect the Trojans.

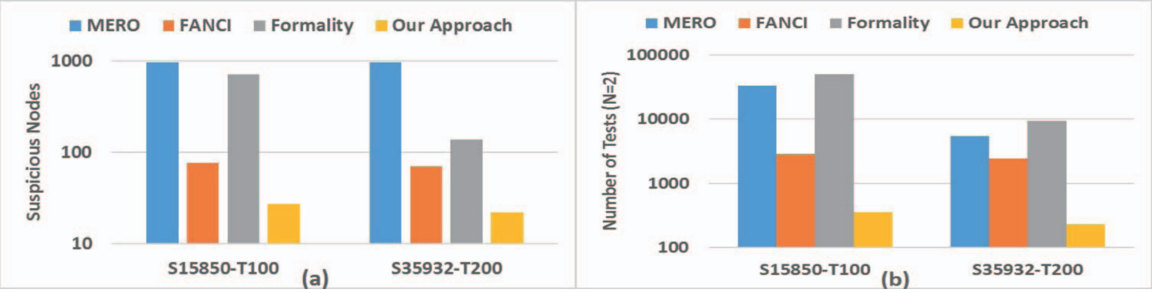

Fig. 5: (a) Number of suspicious nodes, (b) Number of tests needed to activate Trojans

match different points. However, if an adversary inserts a hard-to-detect hardware Trojan, signature analysis may incorrectly match points since their simulation result are same. As a result, *Formality* may not be able to detect inserted Trojans (as indicated in Table I). Our proposed method is based on polynomial manipulation of different regions of the circuit and it is not dependent on the simulation or pattern generation. Thus, our method outperforms *Formality* when there are hard-to-activate Trojan in the implementation.

B. Trojan Localization

Table I presents results for hardware Trojan localization. The first three columns show the type of benchmarks, number of gates in the circuit, and number of malicious gates (consisting of Trojan trigger and payload), respectively. The fourth column shows the number of suspicious gates reported by *"FANCI"* [5] approach. *FANCI* reports 1% to 8% of circuit nodes as false positive nodes on average (we have reported suspicious nodes as false positive nodes plus actual Trojan gates). The fifth columns shows the number of suspected gates that can be found using *Formality*. It reports some faulty flip-flops or primary outputs which may have different values because of change in the functionality. However, there are so many gates in the cone corresponding to the faulty primary outputs or flip-flops and all of these gates are suspicious. In case Vga-lcd-T100, the Trojan effects are masked due to observability issues and nature of the above-mentioned signature analysis, and *Formality* returns no suspicious nodes. The sixth column shows the number of suspicious gates that our method finds. Our method detects all of the Trojan circuit gates (no false negative gates) plus very small number of false positive nodes (benign gates). The seventh column shows the number of specification polynomials which is equal to number of flip-flops in the design plus number of primary outputs. The eighth column presents the number of implementation polynomials which is equal to number of fanout-free cones existing in the implementation. The CPU time (in seconds) to localize the Trojan is reported for each benchmark in ninth column. The time complexity of our method is linear with respect to the number of gates. The tenth, eleventh and twelve columns show the number of false positive gates that our approach, FACNI [5] and *Formality* [22] report, respectively. Clearly, our approach returns only few false positive gates. We are aware of the fact that comparison with FANCI is not fair since it does not requires golden model. However, FANCI returns a lot of suspicious gates that it may not include all of the Trojan gates. For example, FANCI has reported top twenty suspicious gates for S35932-T200, none of them are from Trojan gates. Moreover, FANCI returns a set of suspicious gates even when the circuit is Trojan free. The next columns show our improvement in comparison with *FANCI* and *Formality* based on number of false positive gates. Our approach has a significant improvement compared to existing approaches - our approach reports orders-of-magnitude less false positive gates compared to [5] and [22].

C. Test Generation

For test generation, we used Tetramax [25], the ATPG tool from Synopsys to generate tests exhaustively to activate the reported suspicious nodes. Since our suspicious candidates are few, we can exhaustively check several combinations to activate the Trojan. However, without using our localization method or using heuristic methods such as [5], exhaustive method will not work due to large number of suspicious gates. Table II shows the number of tests needed for activation and detection of Trojans with/without using our localization method. First column shows the type of benchmark (same as Table I). The next two columns present the number of required tests to activate trigger conditions one at a time without and with using our localization method, respectively. The next column shows our improvement compared to without using localization. Our proposed approach improves number of required test vectors significantly. The next columns show the number of required tests to activate trigger conditions of two and four nodes at a time without and with using our localization method and the associated improvements, respectively. As it can be seen from Table II, it is impractical to generate tests to activate four-node triggers even for these small benchmarks without our localization approach. If our localization is utilized, the number of required tests are reasonable and would be less by several orders of magnitude.

TABLE II: The required tests to activate the Trojan

Benchmark	N=1			N=2			N=4		
	W/O Localization	With Localization	Improvement	W/O Localization	With Localization	Improvement	W/O Localization	With Localization	Improvement
RS232-T1000	311	13	23.9x	48205	78	618.0x	4E+8	715	5E+5x
RS232-T1100	310	14	22.1x	47895	91	526.3x	4E+8	1001	4E+5x
S15850-T100	2456	27	91.0x	3E+6	351	8.6E+3x	2E+12	17550	9E+7x
S38417-T200	5823	26	224.0x	2E+7	325	5.2E+4x	5E+13	14950	3E+9x
S35932-T200	5445	22	247.5x	1E+7	231	6.4E+4x	4E+13	7315	5E+9x
S38584-T100	7580	11	689.1x	3E+7	55	5.2E+5x	1E+14	330	4E+11x
Vga-lcd-T100	70162	22	3189.2x	2E+9	231	1.1E+7x	1E+18	7315	1E+14x
Average	13155.28	19.85	640.97x	2.9E+08	194.57	1.6E+6x	1.4E+17	7025.14	1.4E+13x

We also compared with MERO [1] for benchmarks S15850-T100 and S95932-T200. We did not compare using the remaining benchmarks because [1] did not report data for those benchmarks. Figure 5(a) shows the number of suspicious gates reported by our approach compared to MERO. Clearly, our approach provides up to 44 times (40 times on average) reduction in suspicious gates compared to MERO. Figure 5(b) compares the number of tests required to activate the Trojan. As shown in the figure, our approach requires up to two orders of magnitude (60 times on average) less test vectors compare to MERO.

The experimental results demonstrate four important aspects of our approach. First, the number of false positive gates are very small and in some cases there are no false positives. In these cases, our method is able to detect the whole Trojan circuit. Next, all of the Trojan payload and trigger gates are inside the list of suspicious gates. In other words, our approach does not produce any false negative result. Our approach detects both sequential and combinational Trojan circuits. Finally, our approach generates very few suspicious nodes (less than 0.2% of original design, less than 0.03% in most cases) that enables us to exhaustively generate tests to activate various trigger conditions to detect the Trojan circuit.

VI. Conclusion

In this paper, we presented an automated approach to localize functional Trojans in third-party IPs. First, we identified whether a third-party IP contains malicious functionality or it is trustworthy. Next, we presented an algorithm to localize the suspicious area of the Trojan-inserted IP to a region which contains very few (less than 0.03% of the original design in most cases) gates. Our approach does not require any unrolling or simulation of the design and it formally identifies the parts of the circuit that is Trojan free as well as the remaining suspicious gates. In order to further aid in Trojan detection, we proposed a greedy test generation method to activate the Trojan. Our experimental results demonstrated the effectiveness of the proposed methodology on trust-HUB benchmarks. Our localization approach reduces the overall Trojan detection effort (number of tests) by several orders of magnitude compared to the existing state-of-the art techniques.

VII. Acknowledgments

This work was partially supported by grants from NSF (CNS-1441667), SRC (2014-TS-2554) and Cisco.

References

[1] R. S. Chakraborty, F. Wolf, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)*, 2009, pp. 369–410.

[2] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *Hardware-Oriented Security and Trust (HOST)*, 2008, pp. 51–57.

[3] J. Aarestad, D. Acharyya, R. Rad, and J. Plusquellic, "Detecting trojans through leakage current analysis using multiple supply pad I_{ddq}s," in *IEEE Transactions on Information Forensics and Security*, 2010, pp. 893–904.

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

[5] A. Waksman, M. Suozzo, and S. Sethumadhavan, "Fanci: Identification of stealthy malicious logic using boolean functional analysis," in *ACM SIGSAC Conference on Computer &#38; Communications Security*, 2013, pp. 697–708.

[6] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa, "A score-based classification method for identifying hardware-trojans at gate-level netlists," in *Design Automation and Test in Europe(DATE)*, 2015, pp. 465–470.

[7] B. Çakir and S. Malik, "Hardware trojan detection for gate-level ics using signal correlation based clustering," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 471–476.

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

[9] *Trust-HUB*, https://www.trust-hub.org/.

[10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation,," in *IEEE Transactions on Computers*, 1986, pp. 677–691.

[11] M. J. Ciesielski, C. Yu, W. Brown, D. Liu and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," in *IEEE/ACM International Conference on Computer Design Automation(DAC)*, 2015, pp. 1–6.

[12] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[13] S. Saha, R. Chakraborty, S. Nuthakki, Anshul, and D. Mukhopadhyay, "Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2015, pp. 577–596.

[14] Y. Huang, S. Bhunia, and P. Mishra, "Mers: Statistical test generation for side-channel analysis based trojan detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 130–141.

[15] S. Narasimhan, X. Wang, D. Du, R. Chakraborty, and S. Bhunia, "Tesr: A robust temporal self-referencing approach for hardware trojan detection," in *Hardware-Oriented Security and Trust (HOST)*, 2011, pp. 71–74.

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

[17] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *HOST*, 2011, pp. 67–70.

[18] J. Rajendran, V. Vedula and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *ACM/IEEE Design Automation Conference (DAC)*, 2015, pp. 112–118.

[19] F. Farahmandi and B. Alizadeh, "Grobner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," in *Microprocessor and Microsystems - Embedded Hardware Design*, 2015, pp. 83–96.

[20] F. Farahmandi, B. Alizadeh, and Z. Navabi, "Effective combination of algebraic techniques and decision diagrams to formally verify large arithmetic circuits," in *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2014, pp. 338–343.

[21] F. Farahmandi, P. Mishra, "Automated test generation for debugging arithmetic circuits," in *Design Automation and Test in Europe Conference(DATE)*, 2016, pp. 1–6.

[22] *Synopsys, Formality*, http://www.synopsys.com/Tools/Verification/ FormalEquivalence/Pages/Formality.aspx, 2015.

[23] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken and R. Drechsler, "Formal verification of integer multipliers bycombining gröbner basis with logic reduction," in *Design Automation and Test in Europe Conference(DATE)*, 2016, pp. 1–6.

[24] *Formality, User Guide*, http://www.vlsiip.com/formality/ug.pdf, 2007.

[25] *Synopsys, Tetramax ATPG*, http://www.synopsys.com/Tools/Implementation/ RTLSynthesis/Test/Pages/TetraMAXATPG.aspx.