HaTCh: A Formal Framework of Hardware Trojan Design and Detection

Syed Kamran Haider[†], Chenglu Jin[†], Masab Ahmad[†], Devu Manikantan Shila[‡], Omer Khan[†] and Marten van Dijk[†]

[†]University of Connecticut –
{syed.haider, chenglu.jin, masab.ahmad, khan}@uconn.edu, vandijk@engr.uconn.edu
[‡]United Technologies Research Center – manikad@utrc.utc.com

May 31, 2015

Abstract

Use of third party 'closed source' IP cores has become a common practice in Electronic Design Automation (EDA) industry. However, these closed source IP cores can potentially contain hardware trojans. Since a closed source IP core is usually provided as a generic gate level netlist which is then instantiated in millions of chips, the adversary can exploit this scalability to infect millions of chips. Therefore, the first observation is that the trojans must be detected in pre-silicon phase; typically done through logic testing.

Moreover, existing tools for hardware trojan detection claim to have a certain level of security by guaranteeing a certain (small) false negative rate for publicly available benchmarks. This implies that only this small *constant* set of benchmarks can be detected with zero (or small) false negative rate. Since an adversary can always create a new trojan which bypasses the detection tool tested on the small constant set of trojan benchmarks, a rigorous security framework of hardware trojans should characterize the potentially *exponentially large* class of hardware trojans that a tool can detect with negligible false negative rate.

We present HaTCh, a first rigorous framework of hardware trojan design and detection within the paradigm of pre-silicon logic testing based tools. We first notice that for the group of non-deterministic hardware trojans/IP cores, no (logic testing based) tool exists that, given a security parameter λ, can detect all trojans in this group with overwhelming probability $1 - negl(\lambda)$. Then we propose, for the other (exponentially large) group of deterministic trojans/IP cores, a detection algorithm which detects *any* hardware trojan from that group with overwhelming probability $1 - negl(\lambda)$. If certain global characteristics regarding the stealthiness of such a hardware trojan are known, then detection becomes polynomial in the number of wires of the IP core. We implemented this algorithm and tested it on existing trojan benchmarks and also on a newly designed advanced trojan.

Keywords: Hardware Trojans, Security, IP Cores

Contents

1 Introduction

Modern electronic systems heavily use third party IP (intellectual property) cores as their basic building blocks. Optimized for area and performance, the IP cores are essential elements of design reuse in electronic design automation (EDA) industry; compared to redesigning components from scratch, this saves a lot of resources.

IP cores are broadly categorized as hard IPs which are transistor level representations of the core, and soft IPs that are offered either in a hardware description language (e.g., Verilog or VHDL) as synthesizable RTL or as generic gate-level netlists. Soft IP cores offered as gate-level netlists are usually called 'closed source', as their high level RTL source code is not provided. Such cores give their vendors significant protection against reverse engineering of the cores as they obfuscate algorithmic and implementation tricks.[1]

Threat of Hardware Trojans: Third party closed source IP cores give rise to a critical security problem: how to make sure that the IP core does not contain a *Hardware Trojan*? A (compromised) IP core vendor acting as an adversary could implant a malicious circuitry in the IP core for privacy leakage or denial of service attacks [3].

The IP core netlists are used as black box modules in larger designs based on which millions of chips are then fabricated. This scalability motivates the adversary to supply an infected IP core resulting in millions of infected chips. Therefore, IP cores should be tested for hardware trojans in pre-silicon phase, i.e. before integration into a larger design and fabricating it. Logic or functional testing, used by Design for Test (DFT) community for testing basic manufacturing defects, is one of the simplest methods to test the IP core for basic hardware trojans which can be easily implemented using the existing simulation/testing tools. For the above reasons we restrict ourselves to analyzing logic testing based tools used in pre-silicon phase for trojan detection.

Software vs. Hardware Trojans: Hardware trojans can be thought of as analogous to software trojans or malwares which infect software applications for similar malicious purposes as hardware trojans [4]. Software trojans can be dynamically injected into the application code and can change their software footprint on the fly, and hence they are difficult to detect. One may think that hardware trojans are also hard to detect like software trojans; however, this is not true. Hardware trojans are static in nature because they are *built* into the hardware and therefore cannot be 'injected' into the IP core on the fly.

False Negatives – A Misleading Metric: A significant amount of research has been done to design efficient tools for hardware trojan detection. These tools are tested on a small constant set of publicly available benchmarked hardware trojans such as TrustHub [5]. The level of security of these tools is reported in terms of the false negative rate observed for the tested benchmarks, where a false negative represents a scenario when a benchmarked trojan is not detected by the tool. Such false negative rate provides misleading information about the effectiveness of the detection tool. Even a 0% false negative rate would provide guaranteed detection only for the small set of tested benchmarked trojans (e.g. TrustHub has only less than 100 benchmarks in total). Whereas in the real world, an adversary may design a new trojan which is different from the tested benchmarks in that it bypasses the detection tool. Therefore, a rigorous security framework of hardware trojans should characterize the potentially *exponentially large* class of hardware trojans from which trojans can be detected (with possibly an exponential amount of work) with negligible false negative rate;

[1]To even improve the strength of obfuscation in the near future, indistinguishability obfuscators $i\mathcal{O}$ (for polynomial-size circuits), as recently developed in the crypto community, may be used as soon as their constructions attain acceptable performance overheads [1] [2].

detection tools should be evaluated based on the subclass of hardware trojans that they can detect with acceptable performance overhead.

A Rigorous Framework – HaTCh: In this paper, we present a rigorous framework called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh) for hardware trojan design and detection. We assume a model where the IP core is closed source; hence, the IP core vendor only provides the generic gate level netlist of the IP core, and its functional specifications in the form of a polynomial time algorithm which can be used to verify the I/O behavior of the core. A precise definition of the hardware trojan detection problem for a group of hardware trojans H is as follows:

Definition 1. *We define the trojan detection problem for H as the problem of designing a detection algorithm (tool) which satisfies the following input/output requirements:*

Inputs:

- *IP core netlist & functional specifications (i.e. the algorithm is a logic testing based tool)*
- *Security parameter λ*
- *Maximum acceptable false positive rate ρ*

Output:

- *Description of additional (tagging) circuitry (to be added to the IP core in pre-silicon phase) which:*
  - *Detects explicit malicious behavior with probability $1 - negl(\lambda)$*
  - *Offers a false positive rate $\leq \rho$*
  - *Has area overhead polynomial in the IP core size*

In order for an adversary to be able to access embedded hardware trojans in millions of fabricated chips, we assume in this paper only remote adversaries who do not have physical access to exploitable side channels.[2]

In order to solve the hardware trojan detection problem, we first differentiate two groups of IP cores with hardware trojans that do not exploit side channels; H_D and H_{ND}. For trojans $\in H_D$ the IP core expresses deterministic (non-probabilistic) functionality and the IP core spec can be used to verify this functionality. We note, if an IP core $\in H_{ND}$, i.e. its specification allows probabilistic fluctuations in the output (a covert channel is possible), then a logic testing based tool cannot detect a hardware trojan which embeds information at a non-negligible rate within those fluctuations (e.g. by using watermarking techniques). This implies that the trojan detection problem for H_{ND} has no solution.

For the deterministic group H_D, we introduce in section 2.2 certain crucial parameters (t, α, d) of trojans which are related to the stealthiness of hardware trojan functionality. We demonstrate that logic testing based traditional detection tools can be bypassed by trojans that are very "stealthy" (e.g., because of a large d value) and we show that currently benchmarked hardware trojans (having small d) are the simplest ones at the huge trojans landscape, and hence represent just the tip of the iceberg. In particular we introduce a new stealthy trojan, coined the *XOR-LFSR* trojan, which

[2]First, physical presence is not a scalable option for being able to attack (a large subset out of) millions of chips. Second, the timing side channel is the only side channel accessible by a remote adversary: we assume that internet connections add too much timing noise to make use of such channels. Hardware trojans in processors may make use of the cache side channel, which we regard in our framework as a covert channel that can be detected by a logic testing based tool (in Definition 1) as a spec violation of execution times.

cannot be efficiently detected by ordinary means (design knowledge of the trojan itself needs to be incorporated in the detection tool).

We show in section 4 that the trojan detection problem for H_D can be solved: Let n be the number of wires in an IP core (which is potentially infected by a hardware trojan), and let ρ be the maximum acceptable false positive rate of a trojan detection tool. Then

Theorem 1. *There exists a tool, coined HaTCh, that solves the trojan detection problem for the set of trojans $\in H_D$ corresponding to fixed parameters (t, α, d). The computational complexity of HaTCh is $O\left(\frac{\lambda}{\log_2(1/\alpha)} \cdot \frac{(2n^2)^d}{\rho}\right)$ (polynomial in the IP core size n).*

Since d can be as large as n, if d is unknown and we need to use HaTCh for all possible d, then the aggregate computational complexity of HaTCh is exponential in n. Also, if d is only known to be upper bounded by a large constant, the complexity may become impractical: even though HaTCh needs to be used only once during the pre-silicon phase in order to protect millions of chips, a distributed computation of HaTCh may still take too long and can be costly. Understanding the HaTCh framework can be used to design very stealthy hardware trojans; in this paper we design and analyse one such example, called the *XOR-LFSR* trojan.

We have implemented HaTCh and our experimental results demonstrate that:

- HaTCh can detect all publicly available H_D trojans from TrustHub as well as (with large complexity) our own very stealthy *XOR-LFSR* trojan.
- It has low area overhead (on average 4.18% for non-pipelined tagging circuitry for a set of tested benchmarks).
- We introduce another parameter l called locality: knowledge of small l can be used to reduce the complexity of HaTCh.
- We note that HaTCh is also useful in detecting any side channel based or H_{ND} trojan that uses H_D triggering characteristics.

The rest of this paper is organized as follows; Section 2 provides a thorough characterization of hardware trojans based on their different properties, which leads to a clear distinction between two trojan groups H_D and H_{ND}. Section 3 formally defines the IP core and its functional specifications which is used in section 4 to present a detailed implementation of HaTCh. Section 5 shows its experimental evaluation. In order to maintain a smooth flow of the paper for the readers, the existing related work for hardware trojan detection along with its limitations has been presented towards the end of the paper in section 6, and we finally conclude in section 7.

2 Characterization of Trojans

A digital IP core can fall under one of the following three categories based on its level of conformity to the design specifications. It can either contain *a hardware trojan*, or have *an exploit*, or it exhibits *normal behavior*.

A hardware trojan is a malicious *extra* circuitry embedded inside a larger circuit, which results in data leakage or harm to the normal functionality of the circuit once activated. We define *extra* circuitry as redundant logic added to the IP core without which the core can still meet its design specifications[3]. A *Trigger Activated* trojan activates upon some special internal or external event, whereas an *Always Active* trojan remains active all the time to deliver the payload. Once activated, a trojan can deliver its payload either through standard I/O channels or through side channels.

2.0.1 Trigger Condition vs. Trigger Signal

A trigger based hardware trojan usually consists of two parts: a trigger circuitry which activates the trojan upon a rare condition or event called ***trigger condition***, and a payload circuitry which performs the malicious operation called 'payload' as intended by the adversary. The trigger condition manifests itself in the form of a boolean value of certain wires. The trigger circuitry is implemented semantically as a comparator which compares the values of these relevant wires with the desired trigger condition and outputs the result in the form of a boolean value on another wire $Trig$ which we call the ***trigger signal*** (i.e. $Trig = 1$ upon trigger condition, $Trig = 0$ otherwise). The payload circuitry takes the trigger signal $Trig$ as input and performs malicious operation once $Trig$ is asserted. The *trigger signal* must not be confused with *trigger condition*; trigger condition is an event which causes the trojan activation, whereas trigger signal is the output of trigger circuitry which signals the payload circuit to show malicious behavior.

Typically, a malicious activity by a trigger based trojan is a result of the following sequence of events:

1. Occurrence of trigger condition
2. Activation of payload circuit
3. Manifestation of malicious behavior

2.0.2 Explicit vs. Implicit Malicious Behavior

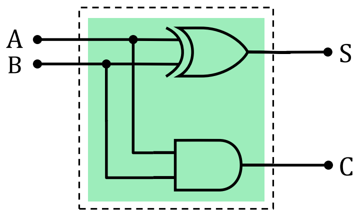
Depending upon the trojan and the IP core, an activated payload circuit behaves in one of the following ways:

1. It causes the core to violate its specifications and this incorrect behavior can be captured at the output channels of the core through logic testing.
2. It forwards such data to the output channels of the core which cannot be distinguished from otherwise 'normal' data and hence the activation of the trojan is not detected through logic testing.

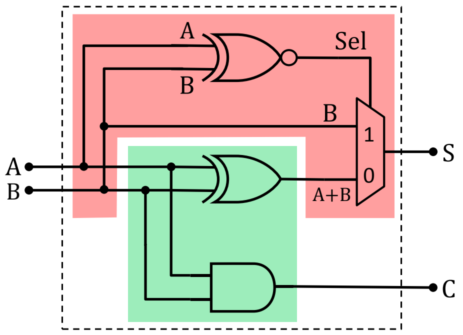
We call these two behaviors as *Explicit* and *Implicit* malicious behaviors respectively. Our proposed detection tool verifies the I/O behavior of the core in a 'learning' phase and if no spec violation (i.e. explicit malicious behavior) is detected, it whitelists the internal states reached by the core. The implicit malicious behavior itself does not harm the normal functionality of the core, however in our model, it may cause some or all of the trigger related wires to get whitelisted during the learning phase because the core does not violate the specs. Consequently the trojan trigger circuitry could be left untracked and as a result, the implicit malicious behavior may help the explicit malicious behavior to go undetected by HaTCh.

Figure 1 shows an example of a simple hardware trojan embedded in a half adder circuit. The trojan free circuit in Figure 1a generates a sum $S = A \oplus B$ and a carry $C = A \cdot B$. The trojan

[3]Design specifications can also cover the performance requirements of the core, and hence pipeline registers etc. added to the core only for performance reasons can also be considered as 'necessary' to meet the design specifications and will not be counted towards 'extra' circuitry.



(a) Half Adder



(b) Half Adder with Trojan: $S = B$ when $A = B$

Figure 1: Example of a simple Hardware Trojan

circuitry, highlighted in red in Figure 1b, triggers when $A = B$ and produces incorrect results i.e. $S = B$ for $A = B$ and $S = A \oplus B$ for $A \neq B$. Notice the difference between the trigger condition $A = B$, and the trigger signal Sel which only becomes 1 when the trigger condition occurs. Also notice that, the trojan affected circuit produces $S = 1$ for trigger condition $A = B = 1$ which is an explicit malicious behavior since it is distinguishable from otherwise normal output ($S = 0$), however the same circuit produces $S = 0$ for trigger condition $A = B = 0$ which is the same as otherwise normal output and cannot be distinguished, hence leading to implicit malicious behavior.

2.0.3 Hardware Trojans Taxonomy

Now that we have explained some important terminologies related to hardware trojans which we will be using frequently in the rest of the paper, we characterize the hardware trojans based on their fundamental characteristics, which leads to a clear distinction between the deterministic H_D and non-deterministic H_{ND} types.

Hardware trojans are first grouped based on the payload channels they use once activated as shown in the Figure 2. *St* refers to the trojans using only standard I/O channels whereas *Si* represents the trojans which also use side channels to deliver the payload. I/O channels are generally used to communicate binary payloads b_j at certain times t_j for the duration of the execution of the IP core. In this sense the view of an I/O channel can be represented as a sequence $(b_1, t_1), (b_2, t_2), \ldots, (b_N, t_N)$. Its information is decomposed in three channels: the binary channel corresponding to $(b_1, b_2, \ldots, b_N)$, the timing channel corresponding to $(t_1, t_2, \ldots, t_N)$, and the termination channel N which reveals information about the duration of the execution of the IP core. If a trojan delivers some of its payload over the timing channel (or other side channels), then we define it to be in *Si*. If a trojan delivers *all* of its payload using the standard usage of I/O channels (the binary and termination channels), then we define it to be in *St*. E.g., a hardware trojan causing performance degradation in terms of slower response/termination times due to slower computation (denial of service in the most extreme case) is in *St*.

As shown in Figure 2, we further refine our description of *St* trojans by subdividing them in *St-D* and *St-ND* groups based on the IP core behavior in which they are embedded and their algorithmic specifications. *St-D* trojans are the ones which are, (1) embedded in an IP core whose output is a function of only its input – i.e. the logical functionality of the IP core is deterministic, and (2) the

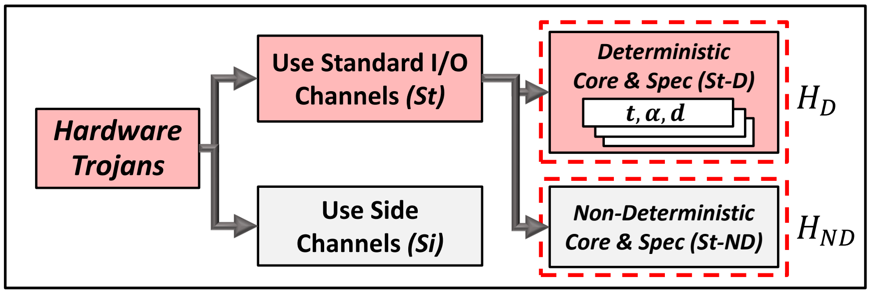

Hardware Trojans
Use Standard I/O Channels (St)
Use Side Channels (Si)
Deterministic Core & Spec (St-D)
t, α, d
Non-Deterministic Core & Spec (St-ND)
H_D
H_{ND}

Figure 2: Classification of Hardware Trojans

algorithmic specification of the IP core can exactly predict the IP core behavior. If any of the two above mentioned conditions are not satisfied then we consider the trojan to be in *St-ND*. A true random number generator (TRNG), for example, is a non-deterministic IP core since its output cannot be predicted and verified by logic testing against an expected output.[4] Any *St* trojan in such a core is considered *St-ND*. A pseudo random number generator (PRNG), on the other hand, is considered a deterministic IP core as its output depends upon the initial seed and is therefore predictable by a logic based testing tool (hence *St-D*). Similarly, if the algorithmic specification allows coin flips generated by a TRNG then we consider the trojan to be *St-ND*. On the other hand, if the coin flips are generated by a PRNG then we regard the trojan as *St-D*.

2.1 Non-Deterministic Trojans H_{ND}

We consider all *St-ND* trojans to be the group of non-deterministic trojans H_{ND}. The non-deterministic behavior of IP cores and/or their functional specification which accepts small probabilistic fluctuations within some acceptable range allows a covert channel for *St-ND* trojans to embed some minimal malicious payload in the standard output without being detected by an external observer. The external observer considers these small fluctuations as part of the functional specification. Hence, the non-deterministic nature of H_{ND} trojans prohibits the development of a logic testing based tool to detect these trojans with overwhelming probability.

E.g., the probabilistic ElGamal signature scheme produces signatures of the form $s = (H(m) - x \cdot (g^k \mod p))k^{-1} \mod (p-1)$, where $k \in \{1, \ldots, p-1\}$ is random with $gcd(k, p-1) = 1$, can be exploited by a hardware trojan in H_{ND} if k is extracted from a TRNG. The secret key x can be leaked as follows. The trojan implements a secret pseudo random permutation R only known to the adversary. The trojan keeps a counter and for the i-th encryption it simply produces a random k with $gcd(k, p-1) = 1$ among those k for which the least significant bit in $R(s)$ is equal to the $(i \mod |x|)$-th bit in the binary representation of x, hence, an adversary who eavesdrops a consecutive sequence of signatures is able to extract x. A legitimate user cannot see any deviation form ElGamal's functional specification (and even if the user suspects an attack of this kind, it cannot be detected by using the signature and secret key because of the pseudo random permutation R). If k is generated by a PRNG, then its seed can be set/programmed and logic testing (as done in HaTCh) can verify produced signatures to correspond to correct pseudo random numbers (which are not generated by a trojan).

[4]Any IP core which contains a TRNG as a module, yet the I/O behavior of the core can still be predicted is considered *St-D*.

2.2 Deterministic Hardware Trojans H_D

We refer to *St-D* trojans as the deterministic group H_D of hardware trojans. In following discussion, we introduce some crucial properties of H_D trojans that characterize their complexity and stealthiness, and explain these properties w.r.t. an advanced H_D trojan example.

2.2.1 Trigger Signal Dimension 'd'

As explained in section 2.0.1, a trigger signal $Trig$ of a trojan takes a certain value *only* when its corresponding trigger condition occurs. Once a trigger signal occurs, *regardless of the other subsequent user interactions*, the trojan manifests malicious behavior in the form of a payload that violates the functional specification (H_D trojans are defined to only exhibit this kind of malicious behavior). A trigger signal/state $Trig$ is represented as a labeled binary vector representing one or more wires/registers/flip-flops (each carrying a 0 or 1). A set of trigger states $\mathcal{T}$ *represents* the trojan if any of its malicious behavior must have (at some clock cycle) passed through a state in $\mathcal{T}$. The dimension of a set of trigger states $\mathcal{T}$ is defined as $d(\mathcal{T}) = \max_{Trig \in \mathcal{T}} |Trig|$.

Clearly, it becomes difficult to detect trojans which are only represented by sets of trigger states with high dimension d because the set of values that a given trigger signal $Trig$ with $|Trig| = d$ can possibly take grows exponentially in d and only one value out of this set can be related to the occurrence of the corresponding trigger condition and hence used for trojan trigger detection. Clearly, since in theory d can be as large as the number of wires n in the IP core, H_D represents an exponentially (in n) large class of possible hardware trojans.

2.2.2 Payload Propagation Delay 't'

For a set $\mathcal{T}$ which represents a trojan, we know that if the trojan manifests malicious behavior, then it must have transitioned through a trigger state $Trig \in \mathcal{T}$ at some previous clock cycle. Therefore, for sets of trigger states $\mathcal{T}$, we define $t(\mathcal{T})$ as the *maximum* number of clock cycles taken to propagate the malicious behavior after entering a trigger state in $\mathcal{T}$, i.e. from the moment when a trigger condition occurs till the moment when its resulting malicious behavior shows up at the output port.

E.g., consider a counter-based trojan where malicious behavior immediately (during the same clock cycle) appears at the output as soon as a counter reaches a specific value. Then, $t(\{Trig\}) = 0$ for the trigger signal $Trig$ which represents this specific counter value. However, notice that any counter value j clock cycles before reaching the 'specific value' can also be considered as a trigger signal $Trig$ with $t(\{Trig\}) = j$, because eventually after j cycles this $Trig$ manifests the malicious behavior. For any hardware trojan, typically there exists a set of trigger signals which represents the trojan and which has a very small t because of a small number of register(s) between the trigger signal and the output port.

An Advanced H_D Trojan: Figure 3 depicts *k-XOR-LFSR*, a counter based trojan with the counter implemented as an LFSR of size k. The trojan is merged with the circuitry of an IP core which outputs the XOR of k inputs A_j.

Let $\mathbf{r}^i \in \{0,1\}^k$ denote the LFSR register content at clock cycle i represented as a binary vector of length k. Suppose that u is the maximum index for which the linear space L generated by vectors $\mathbf{r}^0, \ldots, \mathbf{r}^{u-1}$ (modulo 2) has dimension $k-1$. Since $dim(L) = k-1 < k = dim(\{0,1\}^k)$, there exists a vector $\mathbf{v} \in \{0,1\}^k$ such that, (1) the inner products $\langle \mathbf{v}, \mathbf{r}^i \rangle = 0$ (modulo 2) for all $0 \leq i \leq u-1$,

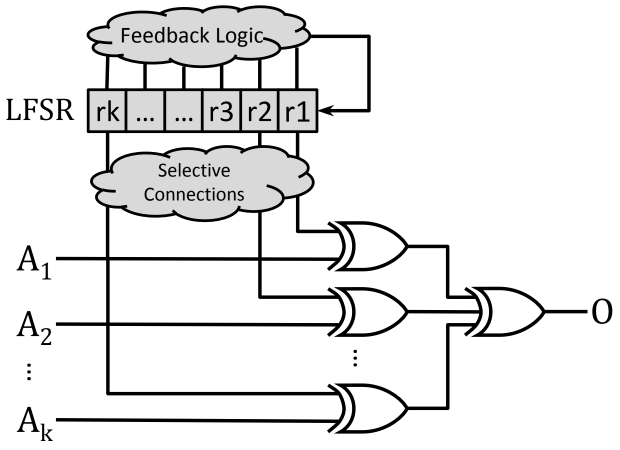


Figure 3: *k-XOR-LFSR*: A general H_D trojan.

and (2) $\langle \mathbf{v}, \mathbf{r}^u \rangle = 1$ (modulo 2). *Only* the register cells corresponding to $\mathbf{v}_j = 1$ are being XORed with inputs A_j.

Since the A_j are all XORed together in the specified logical functionality to produce the sum $\sum_j A_j$, the trojan changes this sum to

$$\sum_j A_j \oplus \sum_{j:\mathbf{v}_j=1} \mathbf{r}^i_j = \sum_j A_j \oplus \langle \mathbf{v}, \mathbf{r}^i \rangle.$$

I.e., the sum remains unchanged until the u-th clock cycle when it is maliciously inverted.

The trojan uses an LFSR to generate register values $\mathbf{r}^i \in \{0,1\}^k$ for each clock cycle i and we assume in our analysis that all vectors $\mathbf{r}^i$ behave like random vectors from a uniform distribution. Then, it is unlikely that u is more than a small constant larger than k (since every new vector $\mathbf{r}^i$ has at least probability 1/2 to increase the dimension by one). Therefore, $u \approx k$, hence, the register size of the trojan is comparable to the number of clock cycles before the trojan is triggered to deliver its malicious payload. This makes the trojan somewhat contrived (since it can possibly be detected by its suspiciously large area overhead).

Since inputs A_j can take on any values, any trigger signal $Trig$ must represent a subset of the LFSR register content. Suppose $t(\{Trig\}) = j$. Then $Trig$ must represent a subset of $\mathbf{r}^{u-j}$. We will proceed with showing a lower bound on $d(\{Trig\})$. Consider a projection P to a subset of d register cells; by $\mathbf{r}|P$ we denote the projection of $\mathbf{r}$ under P, and we call P d-dimensional. If $\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \le i < u-j\}$, then the wire combination of the d wires corresponding to $\mathbf{r}^{u-j}|P$ *cannot* represent $Trig$ (otherwise $t(\{Trig\}) > j$): if this is the case for all d dimensional P, then $Trig$ cannot represent a subset of $\mathbf{r}^{u-j}$. The probability that $\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \le i < u-j\}$ is at least equal to the probability that $\{\mathbf{r}^i|P : 0 \le i < u-j\} = \{0,1\}^d$, which is (by the union bound)

$$\begin{aligned} &\ge 1 - \sum_{w\in\{0,1\}^d} Prob(\{\mathbf{r}^i|P : 0 \le i < u-j\} \subseteq \{0,1\}^d \setminus \{w\}) \\ &= 1 - \sum_{w\in\{0,1\}^d} (1-1/2^d)^{u-j} \approx 1 - 2^d e^{-(u-j)/2^d} \end{aligned}$$

Since there are $\binom{k}{d} \leq k^d/d!$ projections, $Trig$ cannot represent a subset of $\mathbf{r}^{u-j}$ with probability

$$\geq (1 - 2^d e^{-(u-j)/2^d})^{k^d/d!} \tag{1}$$

For $d \geq \log(u-j) - \log(\log(u-j)\log k + \log\log k)$, this lower bound is about $\geq 1/e$. Since $u \approx k$ and after neglecting the term $\log\log k$, this shows an approximate lower bound

$$d(\{Trig\}) \geq \log(k - t(\{Trig\})) - \log(\log(k - t(\{Trig\}))\log k).$$

This characterizes the stealthiness of the *k-XOR-LFSR* trojan.

2.2.3 Stealthiness Factor '$\boldsymbol{\alpha}$'

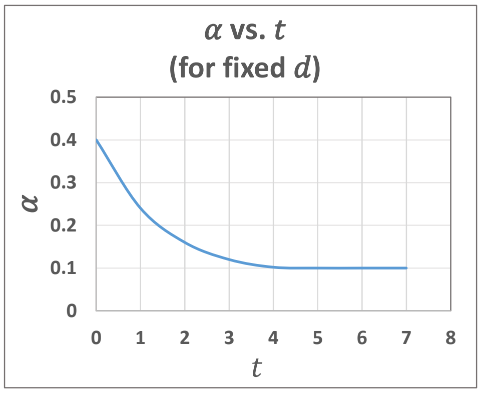
In addition to previously discussed hardware trojan properties, the IP core in which the trojan is embedded plays a critical role in the stealthiness of the trojan. As discussed in section 2.0.2, depending upon the IP core under consideration it may not always be possible to distinguish a malicious behavior from a normal behavior just by monitoring the outputs, i.e. in case of implicit malicious behavior. For example, consider a trojan being a malicious 2-to-1 MUX, with one of its inputs connected to a malicious wire and the other one to a normal wire (as in Figure 1). Suppose the trojan triggers and selects as a result the malicious wire as the MUX output instead of the normal wire. If the normal and malicious input have the same value, then the'malicious' output value is indistinguishable from the normal output value. The effect of the IP core on the stealthiness of a hardware trojan (which is embedded into it) against logic testing techniques can be represented by the stealthiness factor α: For a trigger state $Trig$ we define $\alpha(Trig)$ as the probability given that $Trig$ occurs, this will lead to *implicit* malicious behavior. We define $\alpha(\mathcal{T})$ as the *maximum* of $\alpha(Trig)$ over $Trig \in \mathcal{T}$. We notice that the higher the value of α, the lower the chance of its detection by logic testing and hence the higher the stealthiness and vice versa. (For our *k-XOR-LFSR* trojan, clearly $\alpha(Trig) = 0$ since it *always* produces incorrect output once the trojan gets triggered.)

2.2.4 Achievable Triples (t, α, d)

A hardware trojan can be represented by multiple sets of trigger states $\mathcal{T}$, each having their own t, α, and d values. The collection of corresponding triples (t, α, d) is defined as the achievable region of the hardware trojan. We denote by $H_{t,\alpha,d}$ all H_D type trojans which can be represented by a set of trigger states $\mathcal{T}$ with parameters $t(\mathcal{T}) \leq t$, $\alpha(\mathcal{T}) \leq \alpha$ and $d(\mathcal{T}) \leq d$.

In the remainder of this paper we develop HaTCh which takes parameters t, α and d as input in order to detect hardware trojans from $H_{t,\alpha,d}$. E.g., by taking $t = 0$ we can detect a simple counter-based hardware trojan for small d (as we have seen there exist a trigger state $Trig$ for $t = 0$ in a simple counter-based hardware trojan; HaTCh for $t = 0$ will characterize $Trig$ so that malicious behavior can be prevented). However, for our more complex *k-XOR-LFSR* trojan, HaTCh for $t = 0$ only detects this trojan if d is taken $\geq \log k - 2\log\log k$ which may make HaTCh's computational complexity prohibitive (see Theorem 1).

The choice of parameters t and d significantly affects the stealthiness α of the hardware trojan. Figure 4 shows the effects of changing t and d on the minimum possibkle α in the achievable region of a hardware trojan. Reducing t means that explicit behavior may not have had the chance to occur, hence, the probability α that no explicit behavior is seen increases. Similarly, reducing d



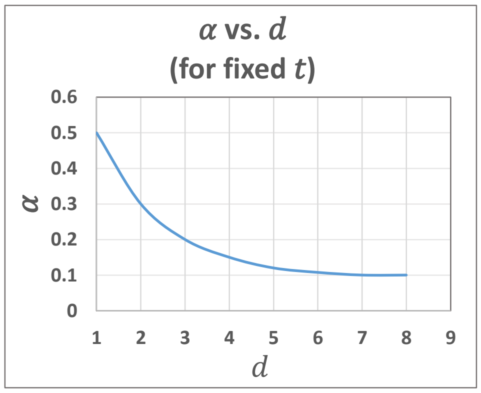


Figure 4: Effects of changing t and d on α.

can increase α since as a result of smaller d, there may not exist a set of trigger signals $\mathcal{T}$ that represents the hardware trojan and satisfies $d(\mathcal{T}) \leq d$. Increasing t or d only decreases α down to a certain level; the remaining component of α represents the inherent implicit malicious behavior of the core.

3 IP Core & Functional Specs

In order to formally model and define hardware trojans, we will first provide a relaxed model for the input and output behavior of the IP cores.

3.1 IP Core

An IP core *'Core'* given as a gate-level netlist represents a circuit module $M = M^{Core}$ (with feedback loops, internal registers with dynamically evolving content, etc.) that receives inputs (over a set of input wires) and produces outputs (over a set of output wires). We define the *state* of M at a specific moment in time (measured in cycles) as the vector of binary values on each wire inside M together with the values stored in each register/flip-flop. Here, the definition of state goes beyond just the values stored in the registers inside M: M itself may not even have registers that store state, M's state is a snapshot in time of M's combinatorial logic (which evolves over time). By S_i we denote M's state at clock cycle i.

User-Core Interaction: We model a user as a polynomial time (pt) algorithm[5] $User$ which, based on previously generated inputs and received outputs, constructs new input that is received by the IP core in the form of a new value. We assume (malicious) users who, due to (network) latencies, cannot observe detailed timing information (a remote adversary can covertly leak privacy over the timing channel if detailed timing information can be observed, which is out of scope of this model). In our model, we only consider trojans that can only deliver a malicious payload over the standard I/O channels in order to violate the functional specification of the core. This implies that only the message contents and the order in which messages are exchanged between the core and user are of importance.

[5] Any random coin flips necessary are stored as a common reference string in the algorithm itself.

We model this by restricting $User$ to a pt algorithm with two alternating modes; an *input generating mode* and a *listening mode*. During the jth input generating mode, some input message X_j is generated which is translated to a sequence $(x_k, x_{k+1}, \ldots, x_n)$ of input vectors for each clock cycle to the circuit module M which defines the IP core. During the jth listening mode of say Δ_j clock cycles, $User$ collects an output message Y_j that efficiently represents the sequence of output vectors $(y_g, y_{g+1}, \ldots, y_k, y_{k+1}, \ldots, y_n, \ldots, y_{n+\Delta_j})$ as generated by M during clock cycles from the end of the last input generating mode at clock cycle g onwards, i.e., the output generated during clock cycles $g, g+1, \ldots, n+\Delta_j$ (here, we write $x_i = \epsilon$ or $y_i = \epsilon$ if no input vector is given, i.e. input wires are undriven, or no output vector is produced). In other words, *User* simply produces an input message X_j, waits to receive an output message Y_j, produces a new input message X_{j+1} and waits for the new output message Y_{j+1} etc. The X_j are produced as semantic units of input that arrive over several clock cycles at the IP core. Y_j concatenates all the meaningful ($\neq \epsilon$) output vectors that were generated by the IP core since the transmission of X_j. This means that the view of the user is simply an ordered sequence of values devoid of any fine grained clock cycle information.

3.2 Functional Specifications

We assume that the IP core has an algorithmic functional specification consisting of two algorithms: *CoreSim* and *OutSpec*. *CoreSim* is an algorithm that simulates the IP core at the coarse grain level of semantic output and input units:

- *CoreSim* starts in an initial state S'_0
- $(Y'_j, S'_j, \Delta_j) \longleftarrow CoreSim(X_j, S'_{j-1})$

CoreSim should be such that it does not reveal any information about how the IP core implements its functionality. It protects the intellectual property (implementation and algorithmic tricks etc.) of the IP core and only provides a specification of its functional behavior. States S'_j are not related to the states S_i that are snapshots of the circuit module M as represented by $Core$. States S'_j represent the working memory of the algorithm *CoreSim*. Notice that *CoreSim* also outputs Δ_j, the listening time needed to receive Y_j if a user would interact with M^{Core} instead of *CoreSim*.

The output specification *OutSpec* specifies which standard output channels should be used and how they should be used. Standard output channels are defined as those which can be configured by the hardware itself (by programming reserved registers etc.). E.g., a hardware trojan doubling the Baud rate (by overwriting the register that defines the UART channel) or a hardware trojan which unexpectedly uses the LED channel (by overwriting the register that programs LEDs), as implemented in [6], would violate *OutSpec*. Notice that side channel attacks are defined as attacks which use non-standard output channels and these attacks are not covered by *OutSpec*.

Emulation of $\mathbf{M^{Core}}$: We assume that the *Core*'s gate-level netlist allows the user of the IP core to emulate its fine grained behavior (the state transition and output vector for each clock cycle), i.e., we assume an algorithm *Emulate*:

- *Emulate*[*Core*] starts in an initial state S_0.
- $(y_i, S_i) \longleftarrow Emulate[Core](x_i, S_{i-1})$.

Emulate[*Core*] behaves exactly as the circuit module M corresponding to *Core*, i.e. *Emulate*[*Core*] and M are functionally the same. The main difference is that *Emulate*[*Core*] parses the language in

Algorithm 1 $User$ interacts with $Emulate[Core]$ and verifies functional correctness and outputs the list of all the emulated states of M^{Core}.

```
procedure SIMULATE($Core, User$)
    $g, Y_0, j, States = 1, \epsilon, 1, [\,]$
    $S_0, S'_0 = ResetStateCore, ResetStateSim$
    while $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$ do
        $(Y'_j, S'_j, \Delta_j) \leftarrow CoreSim(X_j, S'_{j-1})$
        $(x_k, \ldots, x_{k+n}) \leftarrow$ SEND$(X_j)$
        $(x_g, \ldots, x_{k-1}) = (\epsilon, \ldots, \epsilon)$
        $(x_{k+n+1}, \ldots, x_{k+n+\Delta_j}) = (\epsilon, \ldots, \epsilon)$
        for $i \leftarrow g, k+n+\Delta_j$ do                          ▷ Emulate
            $(y_i, S_i) \leftarrow Emulate[Core](x_i, S_{i-1})$
            if $y_i \neq \epsilon$ then $Y_j = Y_j || y_i$
            end if
            $Append(States, S_i)$                                   ▷ Update $States$
        end for
        $j, g = j+1,\ k+n+\Delta_j+1$
        if $Y'_j \neq Y_j$ then                                     ▷ Verification
            return ("Trojan-Detected", $\cdot$)
        end if
    end while
    return ("OK", $States$)                                         ▷ All emulated states
end procedure
```

which $Core$ is written: In practice, one can think of $Emulate[Core]$ as any post-synthesis simulation tool, such as Mentor Graphic's ModelSim [7] simulator, which can be used to simulate the provided IP core netlist $Core$. Notice the following properties of such a simulator tool; firstly it does not leak any information about the IP other than described by $Core$ itself and secondly, it is inefficient in terms of (completion time) performance since it performs software based simulation, however it provides fine grained information about the internal state of the IP core at every clock cycle.

Simulation of User-Core Interaction: The user of the IP core is in a unique position to use $Emulate[Core]$ and verify whether its I/O behavior (over standard I/O channels) matches the specification $(CoreSim, OutSpec)$. The verification can be done automatically without human interaction: This will lead to the proposed HaTCh tool which uses (during a *learning phase*) $Emulate[Core]$ to simulate the actual IP core M^{Core} and verifies whether the sequence $(X_1, Y_1, X_2, Y_2, \ldots)$ of input/output messages to/from $User$ matches the output sequence $(Y'_1, Y'_2, \ldots)$ of $CoreSim$ on input $(X_1, X_2, \ldots)$. Algorithm 1 shows a detailed description of this process (U_i indicates the current state or working memory of $User$).

Notice that $User$ in algorithm 1 can be considered as a meta user which runs several test patterns from different individual users one after another to test M^{Core}. This implies that SIMULATE is generic and can be applied to both a non-pipelined as well as a pipelined M^{Core}.

Functional Spec Violation: We consider H_D trojans, therefore, $CoreSim$ is a non-probabilistic algorithm. This means that the output sequence $(Y'_1, Y'_2, \ldots)$ of $CoreSim$ is uniquely defined (and next definitions make sense): We define the input sequence $X_1, X_2, \ldots, X_N$ to not violate the functional spec if it verifies properly in algorithm 1, i.e., if the emulated output (by $Emulate[Core]$) correctly corresponds to the simulated output (by $CoreSim$). If it does not verify properly, then we say that the input sequence $X_1, X_2, \ldots, X_N$ violates the functional spec.

4 Trojan Detection Tool

In order to prove Theorem 1 we present a powerful trojan detection tool, called HaTCh, which uses a *whitelisting* approach to discriminate the trustworthy circuitry of an IP core from its potentially malicious parts. Algorithm 2 shows the operation of HaTCh. In order to disable any trojan in a $Core \in H_{t,\alpha,d} \subseteq H_D$, HaTCh takes the following parameters as input:

1. $Core$: The IP core under test.
2. $\mathcal{U}$: A user distribution with a pt sampling algorithm SAMPLE where each $User \leftarrow$SAMPLE($\mathcal{U}$) is a pt-algorithm.
3. t, d, α: Parameters characterizing the hardware trojan.
4. λ: A security parameter.
5. ρ: Maximum acceptable false positive rate.

HaTCh processes $Core$ in two phases; a *Learning phase* and a *Tagging phase*. The learning phase performs k iterations of functional testing on $Core$ using input test patterns generated by k users from $\mathcal{U}$ and learns k independent blacklists $B_1, B_2, \ldots, B_k$ of unused wire combinations, where k depends upon the desired security level and is a function of α and λ. If $Core$ is found manifesting any explicit malicious behavior during the learning phase then the learning phase is immediately terminated. This produces an error condition and as a result, HaTCh does not execute its tagging phase and simply returns "Trojan-Detected" which indicates that the IP core contains a hardware trojan, and is rejected straightaway in the pre-silicon phase. On the other hand, if no explicit malicious behavior is observed during the learning phase, a union of all individual blacklists B_i produces a final blacklist B. Having a union of multiple independent blacklists minimizes the probability of incorrectly whitelisting (due to implicit malicious behavior) a trigger wire(s) since the trigger wire(s) need to be whitelisted in all k learning phases in order for the trojan to remain undetected. Once the final blacklist B is available, the tagging phase starts. It transforms $Core$ to $Core_{Protected}$ by adding extra logic for each entry in the blacklist such that whenever any of these wires is activated, a special flag will be raised to indicate the activation of a potential hardware trojan. We notice that logic testing based approach for trojan detection is generally pretty straightforward, and has already been proposed in [8]. Here we use such an approach as our proof methodology for HaTCh.

4.1 Learning Phase

We first present a technical definition of t-legitimate states and d-dimensional projections which will help in explaining the process of whitelisting in the learning phase:

Definition 2. *Let* $(w, States) \leftarrow$ SIMULATE$(Core, User)$. *Assuming* $Core$ *is fixed, we define* $W(User) = w$ *and the set of* t*-legitimate states of* $User$ *as:*

$$L_t(User) = \{States[1], \ldots, States[|States| - t]\}$$

(Since SIMULATE *is deterministic,* $L_t(User)$ *and* $W(User)$ *are well-defined.)*

Algorithm 2 HaTCh Algorithm

```
procedure HaTCh($Core, \mathcal{U}, t, d, \alpha, \lambda, \rho$)
    $k = \lceil \frac{\lambda}{log_2(1/\alpha)} \rceil$, $B = \emptyset$
    for all $1 \le i \le k$ do
        $B_i \leftarrow$ Learn($Core, \mathcal{U}, t, d, \rho$)
        if $B_i =$ "Trojan-Detected" then
            return "Trojan-Detected"
        else
            $B = B \cup B_i$
        end if
    end for
    $Core_{Protected} =$ Tag($Core, B$)
    return $Core_{Protected}$
end procedure
```

Definition 3. *We define a vector* $\mathbf{z}$ *projected to index set* P *as* $\mathbf{z}|P = (\mathbf{z}_{i_1}, \mathbf{z}_{i_2}, \ldots, \mathbf{z}_{i_d})$ *where* $P = \{i_1, i_2, \ldots, i_d\}$ *and* $i_1 < i_2 < \cdots < i_d$*. We call* d *the dimension of projection* P *and we define* $\mathcal{P}_d$ *to be the set of all projections of dimension* d*. We define a "set* Z *projected to* $\mathcal{P}_d$*" as*

$$Z|\mathcal{P}_d = \{(P, \mathbf{z}|P) : \mathbf{z} \in Z, P \in \mathcal{P}_d\}.$$

Formally, a trigger state is a labelled binary vector, i.e., it is a pair $(P, \mathbf{x})$ where P denotes a projection and $\mathbf{x}$ is a binary vector; if $Core$ is in state $\mathbf{z}$ and $\mathbf{z}|P = \mathbf{x}$ then the trojan gets triggered. Now let $\mathcal{T}$ be a set of trigger states/signals which *represents* the hardware trojan, i.e., M^{Core} manifests malicious behavior if and only if it has passed through a state in $\mathcal{T}$. Let $\mathcal{T}$ have dimension d and payload propagation delay t, i.e., the trojan always manifests malicious behavior within t clock cycles after "it gets triggered" by a trigger signal in $\mathcal{T}$. Then we know that a state in $L_t(User)|\mathcal{P}_d$ can only correspond to a trigger signal in $\mathcal{T}$ if the trigger signal produced implicit malicious behavior, i.e., $W(User) =$ "OK". Now we are ready to define $H_{t,\alpha,d}$:

Definition 4. *$Core \in H_{t,\alpha,d}$ if and only if it is represented by a set of trigger states $\mathcal{T}$ with $t(\mathcal{T}) \le t$ and $d(\mathcal{T}) \le d$ such that*

$\mathcal{C}1$) There exists a $User$ and a state S in the set of all reachable states of M^{Core} such that $S \in \mathcal{T}$. I.e., $Core$ is indeed capable of manifesting malicious behavior.

$\mathcal{C}2$) For all $User$, Simulate*($Core, User$) outputs $W(User)$ such that:*

$$Prob\,(W(User) = \text{"OK"} \mid (L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset) \le \alpha$$

We refer to the minimum α that satisfies $\mathcal{C}2$ for $\mathcal{T}$ as the stealthiness factor $\alpha(\mathcal{T})$ of the hardware trojan (cf. section 2.2.3).

Algorithm[6] 3 describes the operation of a single iteration in HaTCh learning phase (lines 3-10 in Algorithm 2). First a $User$ is sampled from $\mathcal{U}$ and at least $1/\rho$ test patterns generated by $User$

[6] In our complexity analysis we assume white listing happens as soon as possible so that double work in lines 14-16 is avoided.

Algorithm 3 Learning Scheme

```
1: procedure LEARN($Core$, $\mathcal{U}$, $t$, $d$, $\rho$)
2:     if I/O register does not match $OutSpec$ then
3:         return "Trojan-Detected"
4:     else
5:         $B = \mathcal{P}_d \times \{0,1\}^d$, $User \leftarrow$ SAMPLE($\mathcal{U}$)
6:         repeat
7:             $B_{old} = B$
8:             Steps from Algorithm 1 from line 2-3
9:             for $m = 1$ to $1/\rho$ do
10:                $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$
11:                Steps from Algorithm 1 from line 5-18
12:            end for                                  ▷ If not aborted, this yields $States$
13:            for all $P \in \mathcal{P}_d$ do                  ▷ Perform Whitelisting
14:                for all $1 \le i \le |States| - t$ do
15:                    $B = B \setminus \{(P, States[i]|P)\}$
16:                end for
17:            end for
18:        until $|B| \neq |B_{old}|$                    ▷ Until no change in blacklist
19:        return $B$                                       ▷ The Blacklist
20:    end if
21: end procedure
```

are tested on $Core$. All those internal states (wires) which are reached by $Core$ during these tests are whitelisted and the rest of the states (wires) are considered to be the part of blacklist. This process is repeated until the blacklist size does not reduce any further, i.e. until $1/\rho$ consecutive test patterns do not reduce the blacklist anymore. This means that neither a false nor a true positive would have been generated if this blacklist were used for the tagging phase. For this reason ρ becomes, statistically, an upper bound on the false positive rate, i.e. if this blacklist is used for the tagging phase, the mean time between false positives is at least $1/\rho$ inputs.

The final blacklist B generated by the learning phase in algorithm 2 is equal to

$$(\mathcal{P}_d \times \{0,1\}^d) \setminus (L_t(User)|\mathcal{P}_d) \tag{2}$$

for the sampled $User$ in line 5. B contains two types of wire combinations: First the wires specifically related to the hardware trojan circuitry, and second some redundant wires which did not excite during the learning phase either because of insufficient user interactions or because of logical constraints of the design.

4.1.1 Security Guarantees of HaTCh

If HaTCh does not detect a functional spec violation during its learning phase, then the blacklist produced by HaTCh is the union of k independent blacklists corresponding to k independent users $User$ with $W(User)$ = "OK", see (2). If the set of trigger states $\mathcal{T}$ is not a subset of this union, then each of the k blacklists must exclude at least one trigger signal from $\mathcal{T}$ and therefore $(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ for each of the corresponding k users $User$. The probability that both

$W(User)$ = "OK" as well as $(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ is at most α (by Bayes' rule) for a $H_{t,\alpha,d}$ trojan. We conclude that the probability that the set of trigger states $\mathcal{T}$ is not a subset of the blacklist produced by HaTCh is at most $\alpha^k \leq 2^{-\lambda}$. So, the probability that the tagging circuitry will detect all triggers from $\mathcal{T}$ is at least $1 - 2^{-\lambda}$.

4.1.2 Computational Complexity of HaTCh

The computational complexity of HaTCh depends upon λ, α, d, ρ, and $n = |Core|$. HaTCh performs k iterations in total during the learning phase in algorithm 2, where $k = \lceil \lambda / \log_2(1/\alpha) \rceil$. The length of each iteration is determined by the number $|\mathcal{P}_d \times \{0,1\}^d| = \binom{n}{d} 2^d \leq (2n)^d$ of possible triggers of dimension d, where size of the IP core $n = |Core|$, and the desired false positive rate ρ: in the worst case every $1/\rho$ user interactions in an iteration may only reduce the blacklist by one possible trigger, hence, the length of each iteration is $O((2n)^d/\rho)$. Whereas, in each iteration, the search space to find and whitelist the projections from is $|\mathcal{P}_d| = \binom{n}{d} \leq n^d$. Therefore the overall computational complexity of HaTCh is given by:

$$O\left(\frac{\lambda}{\log_2(1/\alpha)} \cdot \frac{(2n^2)^d}{\rho}\right) \tag{3}$$

In order to reduce the computational complexity, we introduce the idea of *locality* in gate level circuits, similar to the region based approach in [8]. We define locality l between two wires as the minimum distance between them in the gate level netlist in terms of basic combinational or sequential logic levels (e.g. logic gates and/or flipflops). A low value of l shows that the corresponding wires are in the close vicinity of each other and vice versa.

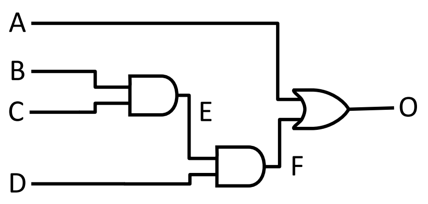
Consider the simple combinational circuit from Figure 5a. Based on this circuit, we draw a *locality graph* shown in Figure 5b whose nodes represent the wires of the circuit and each edge between any two nodes represents connectivity of the corresponding two wires through a combinational logic level. In other words, each logic gate of the circuit is replaced by multiple edges (three in this case) in the graph which connect together the nodes corresponding to its inputs and the output. One can easily find the locality of any two wires in the circuit through its locality graph, e.g. E has locality $l = 1$ with B, C, D and F whereas O has locality $l = 1$ with A and F. On the other hand, E has $l = 2$ with A and O.

Let n_l denotes the maximum number of wires in the locality of any of the n wires of the circuit for parameter l, then the projections search space drastically reduces to $|\mathcal{P}_{d,l}| = n \cdot \binom{n_l}{d-1} \leq n \cdot n_l^{d-1}$ since $n_l \ll n$ Hence, the overall complexity from (3) is reduced to:

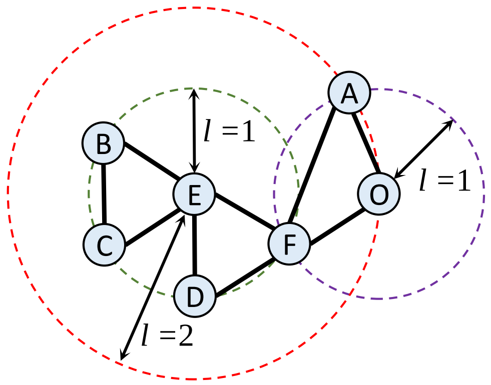
$$O\left(\frac{\lambda}{\log_2(1/\alpha)} \cdot \frac{2n^2(2n_l^2)^{d-1}}{\rho}\right)$$

4.2 Tagging Phase

The tagging phase (i.e. TAG in algorithm 2) takes an untrusted *Core* along with a blacklist B as inputs and adds additional logic to the *Core* to keep track of the suspicious wires in the blacklist. A new output signal called *TrojanDetected* is added to the *Core*. This output is asserted whenever any wire from B takes a 'blacklisted' value. To achieve this functionality, a tree of logic gates is added to *Core* such that the logic 1 is propagated to *TrojanDetected* output



(a) A simple combinational circuit



(b) Locality Graph of the circuit

Figure 5: Example of Locality Graph

whenever a 'blacklisted' value is taken by a suspicious wire. The area overhead of tagging circuitry is $O(|B|d)$ where d represents the parameter passed to HaTCh.

Notice that the added logic can be pipelined to keep it off the critical path and hence it would not affect the design timing. The pipeline resisters may delay the detection of the hardware trojan by $O(\log_2(|B|d))$ cycles, however we show in our evaluation section that for average sized IP cores, HaTCh produces a significantly small B. Consequently, the detection delay because of pipeline registers is only a few cycles. Additionally, for a particular IP core the HaTCh computation needs to be done only once for millions of its instances to be fabricated. Hence, even for larger IP cores, it is worth investing the computational time of several hours to achieve a significantly small blacklist B and to produce millions of trustworthy chips.

5 Evaluation

In this section, we evaluate our HaTCh tool for Trusthub [5] benchmarks. We first analyze the Trusthub benchmarks w.r.t. HaTCh framework. Then we briefly describe our experimental setup and methodology including some crucial optimizations implemented to minimize the area overhead. Finally we present and discuss the experimental results.

5.1 Characterizing Trusthub Benchmarks

Table 1 shows the relevant benchmarks from Trusthub categorized according to the HaTCh framework. *St-D* trojans (i.e. H_D group) is further subdivided based on the properties (t,α,d). All these trojans happen to be represented by a single trigger of dimension $d = 1$ (i.e., the trigger is a single wire); their corresponding t and α values are listed in Table 1.[7] For t values, we simply count the minimum number of registers between the trigger signal wires(s) and the output port of the IP core. In order to estimate α values, we first find the smallest chain of logic gates starting from the trigger signal wire(s) till the output port of the IP core (ignoring any registers in the path). Then for each individual logic gate, we compute the probability of propagating a logic 1 (considering that

[7] α values show estimated upper bounds on probabilities.

Table 1: Classification of Trusthub Benchmarks w.r.t. HaTCh framework

Type		t	α	***Benchmarks***
St	***D***	0	$1/2^{32}$	BasicRSA-T{100, 300}
			0.5	s15850-T100, s38584-T{200, 300}
			0-0.25	wb_conmax-T{100, 200, 300}
			0-0.87	RS232-T{100, 800, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1900, 2000}
		1	0.5	b15-T{300,400}
			0.5-0.75	s35932-T{100, 200}
			0-0.06	RS232-T{400, 500, 600, 700, 900, 901}
		2	0.5	vga-lcd-T100, b15-T{100, 200}
			0.87	s38584-T100
		3	$1/2^{32}$	BasicRSA-T{200, 400}
			0.5	s38417-T100
		5	0.99	s38417-T200
		7	0.5	RS232-T300
		8	0.5	s35932-T300
	ND	N/A		MC8051-T{200, 300, 400, 500, 600, 700, 800}, PIC16F84-T{100, 200, 300, 400}
Si		N/A		AES-T{400, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 2000, 2100}, s38417-T300, AES-T{100, 200, 300}

the trigger wire(s) get a logic 1 upon a trigger event), e.g. an AND gate has the probability 1/4 of propagating a logic 1, whereas an XOR gate has the probability 1/2. Finally we compute an aggregate probability of propagation by multiplying all the probabilities of each logic gate in the chain, which gives the value $1 - \alpha$. HaTCh is able to detect all these *St-D* trojans using $d = 1$.

Notice that all these *St-D* trojans have a very low value of d (particularly $d = 1$) which reflects their low stealthiness, and hence the fact that these publicly available benchmarks represent only a small subset consisting of simple trojans. Even though some of these benchmarks have high values of α (e.g. s38584-T100 and s38417-T200), it is not useful for the adversary to have very high α. Ideally, on one hand, the adversary wants the trojan to be triggered in the learning phase only once, and remain undetected (i.e. by having high α) so that the trojan trigger is whitelisted. On the other hand, after the learning phase, he wants the trojan to deliver the payload by disrupting the normal output (i.e. by having low α), otherwise the trojan is not useful for him. Therefore, the adversary would like to have a sweet spot between the high and low ends of α values. This essentially increases the chances for HaTCh to detect the trojan, i.e. either it gets detected (if triggered) in the learning phase (because $\alpha \ll 1$), or it gets detected by the tagging circuitry later.

St-ND (i.e. H_{ND} group) and *Si* trojans are out of scope of HaTCh. Trigger activated *Si* trojans, however, can still be detected by HaTCh provided that these trojans do not get triggered during the HaTCh learning phase so that their trigger related wires remain blacklisted.

5.2 Experimental Setup & Methodology

We first test five different benchmarks from RS232 and seven from *s-Series* (i.e., s15850, s35932 and s38417) benchmark groups (all of which together form a diverse collection) using the parameters t and α as listed in Table 1. Since these trojans have the dimension $d = 1$, we also set the parameter $d = 1$ for HaTCh. For all our experiments, we set the maximum acceptable false positive rate ρ to be 10^{-5}. HaTCh detects all tested benchmarks, and the resulting area overheads of tagging circuitries are presented in the results section. Notice that s38417-T300 belongs to *Si* type, but since it does not get triggered in the learning phase, HaTCh is still able to detect it.

Even though these benchmarks have a maximum dimension $d = 1$ which means that they can be detected already by using $d = 1$ in HaTCh, we test certain RS232 benchmarks with parameters $d = 2$ and locality $l = 1$ in order to estimate the area overhead for these parameter settings. These results are also presented later in this section.

HaTCh tool works on a synthesized gate level netlist of the IP core. We use Synopsys Design Compiler [9] to synthesize the RTL design. Next, we perform post-synthesis simulations with self checking testbenches using Mentor Graphic's ModelSim [7] simulator. The benchmark is given random test patterns as inputs (ATPG tools can also be used to generate patterns) and the self checking testbench verifies the correct behavior, and the simulation trace of each wire is dumped into a file upon successful verification. HaTCh parses the simulation dump file using an automated script to generate a blacklist. Initially all possible transitions of all the wires of the circuit are blacklisted. Then, every transition read by the script from the simulation file is removed from the blacklist which eventually leads to a final blacklist containing only the untrusted transitions of certain wires. Based on the final blacklist, additional logic is added to flag the blacklisted transitions.

HaTCh tool also performs certain optimizations to remove as much redundant wires from the blacklist as possible. The key idea behind these optimizations is that if the input(s) and output of a logic element coexist in the blacklist, then the output wire can be removed from the blacklist provided that changing the corresponding blacklisted input will affect the output. For example, inverters and logic buffers can benefit from such optimizations. These optimizations lead to a significant reduction in the size of blacklist which in turn reduces the area overhead.

5.3 Experimental Results

Figure 6 shows the size of the blacklists sampled after different numbers of input patterns for s-Series benchmarks. For each benchmark, the blacklist size decreases rapidly with the number of input patterns until it reaches a state when most of the wires in the design are already whitelisted and no more wires are eliminated from the blacklist by further testing. E.g. the blacklists for the s35932 group become stable already after only 100 input patterns. Whereas s38417 group achieves the stable state after 10,000 input patterns. Only s15850 group takes longer to become stable.

Table 2 shows the area overhead incurred by HaTCh for s-Series benchmarks both for non-pipelined and pipelined tagging circuitries. The size of benchmarks (gates+registers) is shown under *Size*. On average, we see an overhead of 8.34% and 4.18% for pipelined and non-pipelined circuitries respectively. For some benchmarks, we see significantly high overhead than others which is most likely because of the fact that the random input test patterns do not provide enough coverage for some of the benchmarks. We observe that the optimizations performed by HaTCh reduce the overheads by ≈ 4.5 times as compared to the un-optimized tagging circuitries. The

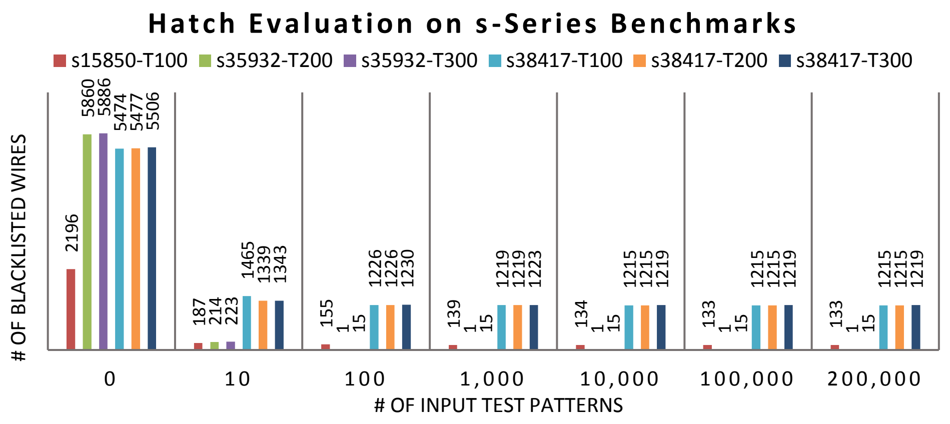


Figure 6: Blacklist size of s-Series with $d = 1$

Table 2: Area Overhead for s-Series with $d = 1$

Benchmark	***Size***	***Area Overhead***	
		Pipelined	***Non-Pipelined***
s15850-T100	2180	4.17%	2.11%
s35932-T200	5442	0.02%	0.02%
s35932-T300	5460	0.16%	0.09%
s38417-T100	5341	15.22%	7.62%
s38417-T200	5344	15.21%	7.62%
s38417-T300	5372	15.25%	7.63%
Average		8.34%	4.18%

Table 3: Area Overhead for RS232 with $d = 2$, $l = 1$

Benchmark	**Size**	**Area Overhead (non-pipelined)**
RS232-T300	280	2.50%
RS232-T1200	273	0.73%
RS232-T1300	267	0.75%

RS232 benchmarks tested with $d = 1$ (i.e. RS232-T{100, 300, 500, 600, 700}) produce blacklists containing only one wire, i.e. the trigger signal. Hence, these benchmarks do not incur any additional area overhead.

Table 3 shows the area overheads of those RS232 benchmarks which we test with dimension $d = 2$ and locality $l = 1$ (i.e. RS232-T{300, 1200, 1300}) in order to get a real estimate of HaTCh overheads for higher dimensions. We see that even with the higher dimension $d = 2$, the overheads for these benchmarks are reasonably small. Since the computed blacklists for these benchmarks are very small, the tagging circuitry would only consist of only 2 to 3 logic levels. Therefore we do not need a pipelined tagging circuitry for these benchmarks.

HaTCh also detects k-*XOR-LFSR* trojan; as an example we implemented it for $k = 4$, and it was detected by using $d = 2$.

6 Related Work

Hardware trojans have recently gained significant interest in the security community [10], [11], [12]. The works [11] and [12] showed how malicious entities can exist in hardware, while Skorobogatov *et al.* [13] showed evidence of such backdoors in military grade devices. Nefarious designs have also been deployed and detected in wireless communications devices [14]. Recent works have mostly focused on detection [15] and identification schemes [16], which assess to what extent the pieces of hardware may be vulnerable, and how related trojans can be classified.

Hicks *et al.* [17] proposed to detect hardware trojans through *unused circuit identification* (UCI). Their solution centers on the fact that the hardware trojan circuitry will not be used within a design, and hence such minimally used logic can be distinguished from the design specification. However, due to functional verification constraints, whole designs cannot be analyzed in optimal time, and hence the scheme identifies large portions of the design as a potential hardware trojan. This results in a high false positive rate, and recent works by some papers have even succeeded in breaking this scheme [18], [19].

Veritrust [20] is another scheme proposed by Zhang *et al.* that identifies suspicious wires that seem redundant in comparison with the design. The scheme uses Karnaugh-maps, and excites portions of the circuit using the design specification, given the fact that the design spec will not activate the hardware trojan. However, the design spec may not activate all circuitry in the design, and the remaining wires are all classified as potential hardware trojans, and contribute heavily to the false positive rate.

Waksman *et al.* presents FANCI which applies boolean function based heuristics to flag suspicious wires in a design [21], stemming ideas from their previous work on hardware obfuscation [22]. This solution may be suitable for cases where backdoors are evident as wires in the design. However, the admitted weakness of this solution is that the scheme suffers from false positives, and is recently broken along with VeriTrust by *DeTrust* [23]. Moreover, this method is a probabilistic method which uses a threshold and some heuristics to determine if a wire should be considered suspicious. This could lead to a false negative where a trojan related wire is regarded as 'not' suspicious because of using a low threshold to reduce false positives.

DeTrust [23] develops new hardware trojans whose circuitries are intermixed with the normal design. Therefore, by having trojan circuits being part of the normal design, previous schemes would designate them as non-malicious, resulting in a false negative rate. However, they only discuss on how to improve the current works (FANCI and VeriTrust) to detect their trojans. The paper also shows that FANCI exhibits a much higher false positive rate than expected.

The above mentioned schemes use trojans from the TrustHub benchmarks suite, in which all trojans are explicitly triggered. This explicitness forgoes the lack of implicitness, due to which all the above schemes are able to detect the benchmarked trojans. These schemes, however, do not cater for higher dimensional ($d > 1$) trojans or the added stealthiness because of the implicit behaviors (i.e. α). DeTrust presents a trojan example which bypasses other existing countermeasures, and interestingly it happens to have the dimension $d = 2$. However, this property has not been noticed or analyzed in that paper. HaTCh fills this gap by providing a detailed and rigorous framework to reason about hardware trojan characteristics and detection schemes.

Further works construct and detect hardware trojans through side channels [24], [25]. Such hardware trojans remain implicitly on, and have no effect on the functionality of the circuit [26]. Side channels include power based channels [27], as well as heat based channels [28]. Power based trojans force the circuit to dissipate more and more power to either damage the circuit, or simply waste energy [29]. Heat based trojans leak important information via heat maps [30], where highs and lows in heat dissipation can be interpreted as 1's and 0's. The presence of a non-zero false negative rate in an adversarial model that allows side channel trojans implies a constant rate of privacy leakage. It is outside the scope of this paper to analyze side-channel models/frameworks in existing literature that may lead to tools that can detect side channel trojans with small false negative rates or obfuscate (by adding extra circuitry) the effect of such trojans leading to reduced privacy leakage rates.

7 Conclusion

We provide the first rigorous framework within which "deterministic trojans", the class H_D, are introduced and analyzed with respect to several stealthiness parameters. We show that currently benchmarked hardware trojans are the simplest ones in terms of stealthiness, and hence represent just the tip of the iceberg at the huge trojans landscape. Based on our framework we were able to design the much more stealthy *XOR-LFSR* hardware trojan. This demonstrates that (1) our framework can be used to understand how to *design stealthy trojans* that force a large complexity overhead for our logic testing based HaTCh tool. This in turn (2) allows us to analyse what kind of additional properties must be satisfied by such very stealthy trojans, and this leads to *counter measures*. E.g., for the *XOR-LFSR* trojan there exists a vector which is orthogonal to the LFSR register before the trojan delivers its payload that violates the functional spec, and also the trojan needs a large register. Both properties can be used to enhance HaTCh in order to efficiently detect an *XOR-LFSR* type trojan.

So far, our best solution of the trojan detection problem for H_D is exponential in the size n of the IP core. When we restrict ourselves to certain subclasses, the solution becomes polynomial in n. Together with the above discussion, this raises the following question: (3) For some $\hat{d}$, does there exist a property, shared among all H_D trojan designs that force HaTCh to have a computational complexity $\geq n^{\hat{d}}$, such that an enhanced HaTCh can be developed which uses this property to detect all H_D trojans with polynomial complexity $O(n^{\hat{d}})$?

We conclude that our HaTCh framework allows the hardware trojan research community to rigorously reason about the effectiveness of different hardware trojans and their existing countermeasures, and also design new and even stronger countermeasures for highly stealthy advanced trojans.

References

[1] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 2013, pp. 40–49.

[2] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Theory of Cryptography*. Springer, 2014, pp. 1–25.

[3] S. Bhasin, J.-L. Danger, S. Guilley, X. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic ip cores," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, Aug 2013, pp. 15–29.

[4] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[5] M. Tehranipoor, R. Karri, F. Koushanfar, and M. Potkonjak, "Trusthub," http://trust-hub.org.

[6] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 2009, pp. 50–57.

[7] "ModelSim, Mentor Graphics Inc." www.mentor.com , http://www.model.com, Wilsonville, OR.

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

[9] "Synopsys Inc." http://www.synopsys.com, Mountain View, CA.

[10] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *Design Test of Computers, IEEE*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[11] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, ser. LEET'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 5:1–5:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387709.1387714

[12] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, no. 5, pp. 34–39, May 2008.

[13] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 23–40.

[14] Y. Liu, Y. Jin, and Y. Makris, "Hardware trojans in wireless cryptographic ics: Silicon demonstration &#38; detection method evaluation," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 399–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561908

[15] T. Reece, D. Limbrick, X. Wang, B. Kiddie, and W. Robinson, "Stealth assessment of hardware trojans in a microcontroller," in *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, Sept 2012, pp. 139–142.

[16] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Provably complete hardware trojan detection using test point insertion," in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, Nov 2012, pp. 569–576.

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

[18] J. Zhang and Q. Xu, "On hardware trojan design and implementation at register-transfer level," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, June 2013, pp. 107–112.

[19] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating uci: Building stealthy and malicious hardware," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 64–77. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.32

[20] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "Veritrust: Verification for hardware trust," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 61:1–61:8. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488808

[21] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 697–708. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516654

[22] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 49–63. [Online]. Available: http://dx.doi.org/10.1109/SP.2011.27

[23] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer & Communications Security*, 2014, *to appear*.

[24] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak, "Hardware trojan horse benchmark via optimal creation and placement of malicious circuitry," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 90–95. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228378

[25] R. Chakraborty and S. Bhunia, "Security against hardware trojan through a novel application of design obfuscation," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, Nov 2009, pp. 113–116.

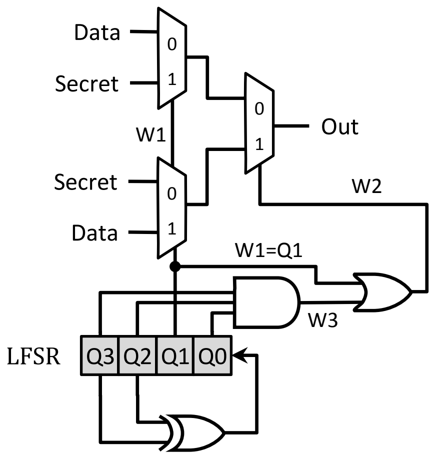
[26] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 396–410.

[27] 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 IEEE International Symposium on*, June 2011, pp. 71–74.

[28] S. Narasimhan, D. Du, R. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia, "Hardware trojan detection by multiple-parameter side-channel analysis," *Computers, IEEE Transactions on*, vol. 62, no. 11, pp. 2183–2195, Nov 2013.

[29] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware trojans," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 632–639. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509456.1509596

[30] D. Forte, C. Bao, and A. Srivastava, "Temperature tracking: An innovative run-time approach for hardware trojan detection," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, Nov 2013, pp. 532–539.



(a) Trojan Circuitry

$Cycle$	$Q3$	$Q2$	$Q1$	$Q0$	$W1$	$W2$	$W3$
0	1	0	1	0	1	1	0
1	0	1	0	1	0	0	0
2	1	0	1	1	1	1	0
3	0	1	1	1	1	1	0
4	1	1	1	1	1	1	1
5	1	1	1	0	1	1	0
6	1	1	0	0	0	0	0
7	1	0	0	0	0	0	0
8	0	0	0	1	0	0	0
9	0	0	1	0	1	1	0
10	0	1	0	0	0	0	0
11	1	0	0	1	0	0	0
12	0	0	1	1	1	1	0
13	0	1	1	0	1	1	0
14	**1**	**1**	**0**	**1**	**0**	**1**	**1**

(b) Truth Table of the Trojan affected circuit. Trigger: $W1 \neq W2$

Figure 7: A Counter-Based $H_{0,0.5,2}$ Trojan to enable the secret leakage

A A Counter-Based $H_{0,0.5,2}$ Trojan Example

The example trojan shown in Figure 7a can leak *Secret* via *Out* port instead of *Data* upon the trigger condition $W1 \neq W2$. The trigger condition is generated by a counter, when reached to (1101), which is implemented as a 4-bit maximal LFSR in order to have maximum possible time before the trojan gets triggered. The LFSR is initialized to $(Q3, Q2, Q1, Q0) = (1010)$ and it can be seen in Figure 7b that if given the parameter $d = 1$, HaTCh will whitelist all the wires related to trigger circuitry only after a few clock cycles since all these wires show transitions. At $14th$ clock cycle, the value of the LFSR becomes (1101) and $W1 \neq W2$, which activates the Trojan to leak the secret.

A.1 Detection by HaTCh with $d = 2$

As it is clear from Figure 7b that, given the parameter $d = 1$, this trojan cannot by detected by HaTCh since all the wires show transitions and get whitelisted after $4th$ clock cycle. Therefore we run HaTCh with a parameter $d = 2$ in order to show that HaTCh is able to detect this trojan. With $d = 2$, HaTCh exhaustively monitors all possible 2-wire combinations of all the wires in the design. It starts with a blacklist of all possible 2-wire combinations (e.g. $\{00, 01, 10, 11\}$) of all the wires in the design and those combinations which are seen during the simulation are removed from the blacklist provided that the output *Out* matches the expected output for every input.

If the learning phase is run for 13 clock cycles, then after the optimizations of HaTCh, we only see one combination of $W1$ and $W3$ in the final blacklist i.e. $(W1, W3) = (0, 1)$ which only occurs upon the trigger condition. All other redundant combinations are optimized away from the blacklist because the logical constraints of the design never allow these combinations to occur in the future, e.g. $(W1, W2) = (1, 0)$ is never possible (unless a stuck at 0 fault for $W2$). Hence HaTCh is able to detect this trojan. Notice that if the learning phase is run for fewer clock cycles, then HaTCh will produce a larger blacklist with more blacklisted combinations.