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

October 5, 2015

Abstract

Electronic Design Automation (EDA) industry heavily reuses existing design blocks called IP cores. These IP cores are vulnerable to insertion of Hardware Trojans (HTs) at design time by third party IP core providers or by malicious insiders in the design team. State of the art research has shown that existing trojan detection techniques which claim to detect all publicly available HT benchmarks, can still be defeated by carefully designing new sophisticated trojans. Researchers have proposed techniques to detect these new trojans, however these techniques are known to be computationally infeasible. This state of affairs leads to two crucial observations. First, instead of guaranteeing a certain (low) false negative rate for a small *constant* set of publicly available benchmarks, a rigorous security framework of HTs should characterize which *exponentially large* class (exponential in number of wires in IP core) of HTs a tool can detect with negligible false negative rate. Second, an effective detection tool must be designed which is computationally feasible for this class of HTs which is orders of magnitude larger compared to the small subclass (e.g. TrustHub) considered in the current literature.

To meet the above mentioned goals, we present HaTCh, the first rigorous framework of HT design and detection within the paradigm of pre-silicon logic testing based tools. We first introduce certain crucial properties of HTs which lead to the characterization of an exponentially large class of HTs that an adversary can (but is not limited to) design, for which we present a detection algorithm which detects *any* HT from this class with overwhelming probability $1 - negl(\lambda)$. Given certain global characteristics regarding the stealthiness of a HT within this class, the computational complexity of our algorithm scales polynomially with the number of wires in the IP core, as opposed to the exponential (in number of IP core inputs) complexity of current state of the art detection schemes to detect such HTs. We have implemented this algorithm, compared it with existing countermeasures, and tested it on TrustHub HT benchmarks, previously proposed HTs which alleviate state of the art detection schemes, and also on a newly designed advanced HT. We argue that those HTs that fall outside the characterized class use HT design principles that allow HTs which can never be detected within the pre-silicon logic testing based paradigm.

Contents

1 Introduction

Modern electronic systems use IP (intellectual property) cores as their basic building blocks. The IP cores are typically offered either in a hardware description language (e.g., Verilog or VHDL) as synthesizable RTL (also called 'open source' IPs) or as generic gate-level netlists (also called 'closed source' IPs). These IP cores give rise to a critical security problem: how to make sure that the IP core does not contain a *Hardware Trojan* (HT)? 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.

IP cores are used as black box modules in larger designs and fabricated inside millions of chips. This scalability motivates the adversary to supply an infected IP core resulting in millions of infected chips. Therefore, IP cores should be tested for HTs 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 HTs 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 HT detection.

A significant amount of research has been done during the past decade to design efficient tools for HT detection. Hicks *et al.* [17] proposed *Unused Circuit Identification* (UCI) which centers on the fact that the HT circuitry mostly remains inactive within a design, and hence such minimally used logic can be distinguished from the other parts of the circuit. However later works [18], [19] showed how to design HTs which can defeat the UCI detection scheme. Zhang *et al.* [20] and Waksman *et al.* [21] proposed detection schemes called VeriTrust and FANCI respectively and showed that they can detect all HTs from the TrustHub [5] benchmark suite. Yet again, the most recent technique called DeTrust [23] introduces new trojan designs which can evade both VeriTrust and FANCI.

This cat-and-mouse game between attackers and defenders leads us to a critical observation, i.e. currently the detection tools are evaluated based on their detection capability over a small *constant* set of publicly available HT benchmarks such as TrustHub. Such evaluation only provides limited information about the effectiveness of the detection tool, because an adversary may design a new HT which is different from the tested benchmarks in that it bypasses the detection tool. Therefore, a rigorous security framework of HTs should characterize the potentially *exponentially large* class of HTs from which HTs can be detected (with possibly an exponential amount of work) with negligible false negative rate. The detection tools should then be evaluated based on the subclass of HTs that they can detect with acceptable performance overhead.

Current state of the art HT detection schemes include extensions of VeriTrust and FANCI, proposed by [23], which enable them to detect somewhat larger subclasses of HTs. The first and admitted limitation of these techniques is their insanely high computational complexity for medium to large sized circuits. Additionally, VeriTrust and many other existing logic testing based HT detection schemes implicitly assume that a HT is either not triggered during the testing phase, or if it does get triggered then it is *always* detected because it produces incorrect output. Surprisingly, there is also a third possibility, i.e. a HT gets triggered during the testing phase yet it does not produce any incorrect output and hence remains undetected. This behavior, termed as *implicit malicious behavior*, leads to serious security vulnerabilities in the current state of the art HT detection schemes such as VeriTrust and extended VeriTrust. We show how an adversary can bypass such detection techniques by exploiting implicit malicious behavior.

1.1 Contributions

In this paper, we present a rigorous framework called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh) for HT design and detection. We assume an adversarial 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. In order for an adversary to be able to access embedded HTs in millions of fabricated chips, we assume in this paper only remote adversaries who do not have physical access to exploitable side channels.[1] The key contributions of the HaTCh framework are as follows:

Characterization of HTs: We first differentiate two groups of IP cores with HTs that do not exploit side channels; H_D and H_{ND}. For HTs $\in H_D$ 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 HT which embeds information at a non-negligible rate within those fluctuations (e.g. by using watermarking techniques).

HT Design Parameters: For the deterministic group H_D, we introduce certain crucial design parameters (t, α, d, l) of HTs which determine the stealthiness of a HT. The impact of having large values of these HT parameters is briefly as follows:

- d: Complicated trigger signal, harder to detect.
- t: Long time until payload triggers, less likely to be detected during testing.
- α: Higher probability of implicit malicious behavior, higher stealthiness.
- l: Weak locality, trigger wires spread out in the circuit, harder to detect.

We demonstrate that logic testing based traditional HT detection tools can be bypassed by "stealthy" HTs (e.g., those with a large d value) and we show that currently benchmarked HTs (having small d) are the simplest ones in the huge HT landscape, and hence represent just the tip of the iceberg. Based on these parameters, we introduce a new stealthy HT, coined the *XOR-LFSR* trojan, which cannot be efficiently detected by ordinary means (design knowledge of the HT itself needs to be incorporated in the detection tool).

HT Detection Tool: We present a powerful HT detection tool for group H_D which solves the following precise HT detection problem:

Definition 1. *HT Detection Problem: Design a logic testing based detection algorithm (tool) for H_D which takes the following inputs:*

- *IP core netlist & functional specifications.*
- *Security parameter λ*
- *Maximum acceptable false positive rate ρ*

and produces the description of additional (tagging) circuitry to be added to the IP core which offers:

[1] 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. HTs 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.

- *Malicious behavior detection probability* $1 - negl(\lambda)$
- *A false positive rate* $\leq \rho$
- *Area overhead polynomial in the IP core size*

We show in section 5 that the HT 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 HT), and let ρ be the maximum acceptable false positive rate of a HT detection tool, then

Theorem 1. *There exists a tool, coined HaTCh, that solves the HT detection problem for the set of HTs* $\in H_D$ *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).*

HaTCh works in two phases; a *learning phase* and a *tagging phase*. The learning phase performs logic testing on the IP Core and produces a blacklist of all the unactivated (and potentially untrusted) transitions of internal wires. The tagging phase adds additional logic to the IP core to track these untrusted transitions. We have implemented HaTCh and our experimental results demonstrate that HaTCh:

- Detects all publicly available H_D trojans from TrustHub as well as (with large complexity) our own stealthy *XOR-LFSR* trojan.
- Incurs low area overhead (on average 4.18% for non-pipelined tagging circuitry for a set of tested HT benchmarks).
- Offers sub-exponential (in number of circuit inputs) computational complexity, in contrast to the exponential complexity of state of the art techniques. E.g. the comparison holds for an m-input pipelined AND gate.

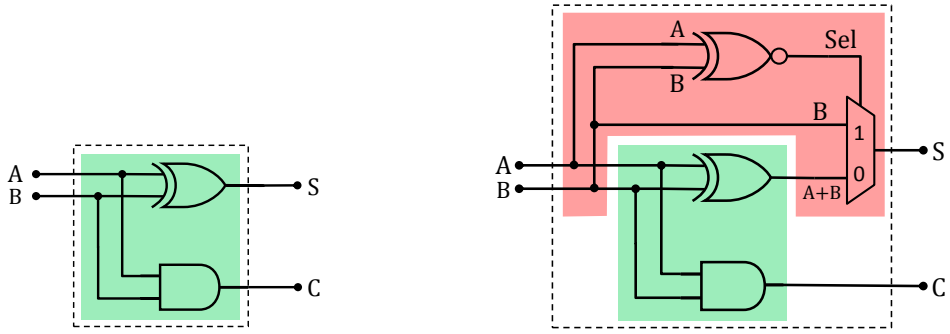
The rest of this paper is organized as follows; Section 2 provides some basic background about the existing HT detection schemes along with their limitations. Section 3 provides a thorough characterization of HTs based on their different properties, which leads to a clear distinction between two trojan groups H_D and H_{ND}. Section 4 formally defines the IP core and its functional specifications which is used in section 5 to present a detailed implementation of HaTCh. Section 6 shows its experimental evaluation and we finally conclude in section 7.

2 Background

A Hardware Trojan (HT) is 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[2]. A *trigger activated* HT activates upon some special event, whereas an *always active* HT remains active all the time to deliver the intended payload.

Trigger activated HTs typically consist of two parts: a *trigger circuitry* and a *payload circuitry*. The trigger circuitry is implemented semantically as a comparator which compares the value(s)

[2]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 HT

Figure 1: A simple HT: Trigger condition $A = B$; Normal output $S = A \oplus B$; Output under trigger condition $S = B$.

of certain wires(s) of the circuit with a specified boolean value called *trigger condition*. The HT trigger circuitry sends its comparator's output to the payload circuitry over certain other wire(s) called the *trigger signal*. Once the trigger signal is asserted, the payload circuitry performs the malicious operation called 'payload' as intended by the adversary.

Definition 2. ***Trigger condition*** *refers to an event, manifested in the form of a particular boolean value of certain internal/external wires of the circuit, which activates the HT trigger circuitry.*

Definition 3. ***Trigger signal or Trigger State*** *refers to a collection of physical wire(s) which the HT trigger circuitry asserts in order to activate the payload circuitry once a trigger condition occurs.*

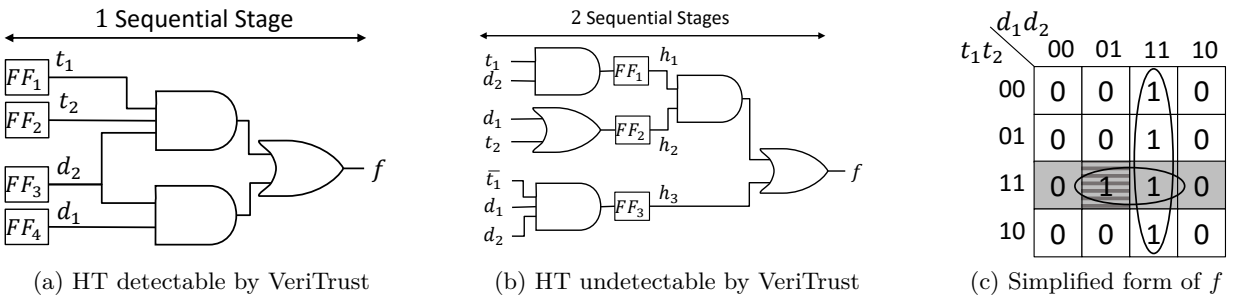
The trigger signal must not be confused with trigger condition; trigger condition is an event which causes the HT activation, whereas trigger signal is the output of trigger circuitry which tells the payload circuitry to show malicious behavior.

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

2.1 Existing HT Detection Techniques

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].

State of the art HT detection schemes are typically based on *trust verification* with the intuition that a HT almost always remains inactive in the circuit to pass the functional verification. *Unused Circuit Identification* (UCI) [17] is one of the first such techniques which distinguishes minimally used logic from the other parts of the circuit. Later techniques have improved over UCI both in terms of security and performance.



(a) HT detectable by VeriTrust (b) HT undetectable by VeriTrust (c) Simplified form of f

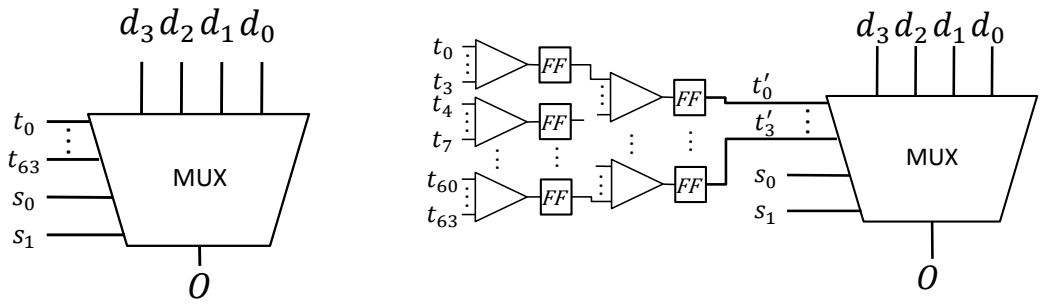
Figure 2: DeTrust defeating VeriTrust: Normal function $f = d_1d_2$, Trojan affected function $f = d_1d_2 + t_1t_2d_2$, Trigger condition $\{t_1, t_2\} = \{1, 1\}$.

VeriTrust: *Veritrust* [20] proposed by Zhang *et al.* detects HTs by identifying the inputs in the combinational logic cone that seem redundant for the normal functionality of the output wire under non-trigger condition. In order to detect the redundant inputs, it first performs functional testing and records the activation history of the inputs in the form of sums-of-products (SOP) and product-of-sums (POS). Then it further analyzes these unactivated SOPs and POSs to find the redundant inputs. However, because of the functional verification constraints, VeriTrust can see several unactivated SOPs and POSs and thus regard the circuit to be potentially infected resulting in false positives.

FANCI: Waksman *et al.* presents FANCI [21] which applies boolean function analysis to flag suspicious wires in a design which have weak input-to-output dependency. A *control value* (CV), which represents the percentage impact of changing an input on the output, is computed for each input in the combinational logic cone of an output wire. If the mean of all the CVs is lower than a threshold, then the resulting output wire is considered malicious. This is a probabilistic method where the threshold is computed with some heuristic to achieve a balance between security and the false positive rate. A very low threshold may result in a high false positive rate by considering most of the wires (even non-malicious ones) as malicious, whereas a high threshold may actually result in false negatives by considering a HT related (malicious) wire to be 'not' malicious.

DeTrust: One of the most recent works *DeTrust* [23] presents a systematic way to design new HTs which cannot be detected by either FANCI or VeriTrust. Notice that, at any time, both VeriTrust and FANCI only monitor the combinational logic between two registers (i.e. one sequential stage). In other words, to decide whether an input to a flipflop (FF) is malicious or not, VeriTrust and FANCI consider the combinational logic cone (i.e. circuitry starting from the outputs of the previous FF stage) driving this wire to be a standalone circuit and then run their respective checks on it. DeTrust exploits this limitation and designs new HTs whose circuitries are intermixed with the normal design and distributed over multiple sequential stages such that FANCI/VeriTrust, while observing a single sequential stage, would consider them non-malicious.

Figure 2 shows an example HT design that DeTrust presents to defeat VeriTrust. DeTrust splits the combinational logic cone of the original (less stealthy) HT design (Figure 2a) into two sequential stages by inserting flipflops (Figure 2b). Since VeriTrust will now consider f to be only a function of h_1, h_2 and h_3; and as none of these inputs are redundant to define f, therefore VeriTrust will identify f to be a non-malicious output. Similarly Figure 3 shows another example HT design



(a) HT detectable by FANCI

(b) HT undetectable by FANCI

Figure 3: DeTrust defeating FANCI: A 4-to-1 MUX is transformed into a malicious MUX with 64 additional inputs where trigger condition is one out of 2^{64} possible input patterns.

by DeTrust which defeats FANCI by spreading the 64 trigger inputs of the original HT design (Figure 3a) over multiple sequential stages (Figure 3b). Hence, even though FANCI/VeriTrust can detect all TrustHub HT benchmarks, DeTrust shows how an adversary can design new HTs to defeat the existing countermeasures.

Extended VeriTrust & FANCI: In order to detect its newly designed HTs, DeTrust proposes extensions of VeriTrust and FANCI. In this paper, we refer to the extended versions of VeriTrust and FANCI as **VeriTrustX** and **FANCIX** respectively. The key idea behind these extended techniques is that the circuits should be monitored up to multiple sequential stages at a time, while ignoring any FFs in between, instead of just one sequential stage at a time. Notice that in the worst case, one may need to monitor all the sequential stages of the design starting from the external inputs. However, if one has some knowledge about the number of sequential stages that a HT can be spread over, VeriTrustX or FANCIX only need to monitor that many number of stages at a time.

For example, VeriTrustX and FANCIX will consider the HTs from Figure 2b and Figure 3b respectively to be one big combinational logic block. Therefore the added stealthiness by the FFs is eliminated and hence both the techniques will be able to detect the respective HTs.

Side-Channel HT Detection Techniques: Further works construct and detect HTs that use side channels [24], [25]. Such HTs remain implicitly on, and have usually no effect on the normal functionality of the circuit. Side channels include power based channels [27], as well as heat based channels [28]. Power based HTs force the circuit to dissipate more and more power to either damage the circuit, or simply waste energy. Heat based HTs leak important information via heat maps [30], where highs and lows in heat dissipation can be interpreted as logic 1's and 0's. The presence of a non-zero false negative rate in an adversarial model that allows side channel HTs 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 HTs with small false negative rates or obfuscate (by adding extra circuitry) the effect of such HTs leading to reduced privacy leakage rates.

2.2 Security Loophole in Existing Techniques

Since VeriTrustX works on the simplified SOP/POS forms of the circuit, both HT designs from Figure 2a and Figure 2b can be represented by the simplified SOP representation shown in Figure 2c. VeriTrust methodology implicitly assumes that a HT is *never* triggered during the functional testing phase, otherwise it would already be detected because of generating the incorrect output. This assumption allows VeriTrust to monitor the activation history of SOPs/POSs rather than each entry in the truth table, resulting in a lower computational complexity. However, it is crucial to note that this assumption is not always correct and it could lead to serious security flaws. Consider the SOP form resulting from Figure 2c, i.e. $f = d_1d_2 + t_1t_2d_2$, the OR of the AND terms d_1d_2 and $t_1t_2d_2$. The possible output values of f which this circuit can produce under the trigger condition $(t_1, t_2) = (1, 1)$ are shaded in gray in Figure 2c. Out of these four 'infected' outputs, the only output which deviates from the 'normal' or 'expected' behavior of the circuit occurs when $(d_1, d_2) = (0, 1)$, and is shaded with horizontal lines in Figure 2c. Rest of the three infected outputs adhere to the design specification of the trojan-free circuit (i.e. corresponding to function $f = d_1d_2$) and therefore can pass the functional testing phase.

If an input pattern $(t_1, t_2, d_1, d_2) = (1, 1, 1, 1)$ is tested on this trojan-affected circuit during the functional testing phase, the resulting output will be $f = 1$. This input does activate the HT yet the functional verifier does not detect any incorrect behavior since $f = 1$ is the expected output corresponding to the trojan-free circuit (i.e. $f = d_1d_2$). Notice, however, that this activation causes the term $t_1t_2d_2$ to be removed from the list of unactivated terms from the SOP form of f, causing the HT to remain undetected by VeriTrust/VeriTrustX. We refer to such behavior of a HT as *implicit malicious behavior*.

Definition 4. ***Implicit malicious behavior*** *refers to a behavior of a HT where the HT generated output is indistinguishable from a normal output.*

Definition 5. ***Explicit malicious behavior*** *refers to a behavior of a HT where the HT generated output is distinguishable from a normal output.*

Once the trojan-affected circuit successfully passes the functional testing phase, it is accepted as a *Trojan-free* circuit which can then freely produce incorrect outputs (i.e. explicit malicious behavior) and harm the normal function of the system or leak sensitive information.

Notice how implicit malicious behavior may misguide the detection tools to consider an infected circuit as a non-infected one, i.e. it leads to a false negative. Also notice that we have a strong definition of false negative, i.e. once a HT-infected circuit is identified as Trojan-free by a detection tool, we count it as a false negative regardless of whether the HT-infected circuit exhibits explicit malicious behavior afterwards at all or not. The intuition behind this strong definition is that even if the HT-infected circuit never exhibits explicit malicious behavior, the implicit malicious behavior can also harm the system by leaking sensitive information. For example, in the above mentioned HT-infected circuit, if an attacker knows $t_1 = t_2 = 1$ and knows $d_1 = 0$ (e.g. d_1 is an input) but does not know d_2 (e.g. d_2 is some secret key bit), then even if $f = 0$ as expected according to normal behavior specified by $f = d_1d_2$ as expected, it leaks $d_2 = 0$ to the adversary.

Definition 6. ***False Negative*** *refers to a scenario when a HT detection tool identifies a circuit containing a HT as a Trojan-free circuit or transforms a circuit containing a HT into a circuit which still allows the HT to express implicit or explicit malicious behavior.*

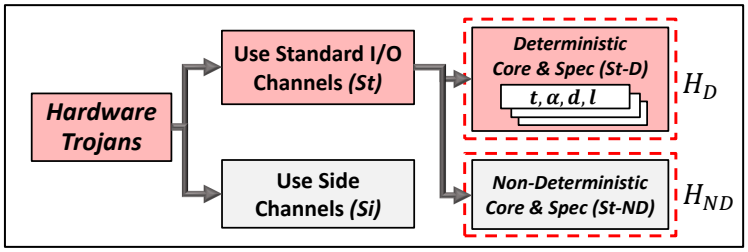


Figure 4: Classification of Hardware Trojans

Clearly, all existing dynamic analysis (i.e. functional testing) based approaches which assume that a HT is never triggered during the functional testing phase can suffer from false negatives because of the implicit malicious behavior. Neglecting the implicit malicious behavior could lead to devastating consequences in a security critical application, e.g. if an adversary designs a HT to significantly increase the probability of implicit malicious behavior and thus alleviates the existing HT detection techniques. Our proposed algorithm HaTCh, on the other hand, does take the implicit malicious behavior of a potential HT into account and eliminates false negatives with overwhelming probability[3].

3 Characterization of Hardware Trojans

In this section, we characterize the HTs based on their fundamental characteristics, which leads to a clear distinction between the deterministic H_D and non-deterministic H_{ND} types.

St vs. Si: Hardware trojans are first grouped based on the payload channels they use once activated as shown in Figure 4. *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*.

St-D vs. St-ND: As shown in Figure 4, 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 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

[3]Since we add tagging circuitry, therefore even if we don't catch the HT during learning phase, the tagging circuitry prevents the final circuit from expressing malicious behavior with overwhelming probability

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*.

3.1 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.

3.2 Deterministic Hardware Trojans H_D

We refer to *St-D* trojans as the deterministic group H_D of hardware trojans. In the 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.

When a trigger condition of a hardware trojan occurs, *regardless of the other subsequent user interactions*, its trigger circuitry gets activated and outputs a certain binary value on a certain trigger signal $Trig$ to activate the payload circuitry which manifests malicious behavior. A trigger signal $Trig$ is represented as a labeled binary vector of one or more wires/registers/flip-flops (each carrying a 0 or 1). In other words, $Trig$ represents a trigger state of the circuit through which the circuit must have passed before manifesting malicious behavior. Notice that the trigger state(s) can be associated back to the occurrence of the respective trigger condition(s). Hence a HT can be represented by a set of trigger states $\mathcal{T}$; i.e. the states which always lead to malicious behavior (implicit or explicit).

Definition 7. *A set $\mathcal{T}$ of trigger states* **represents** *a HT if the HT always passes through one of the states in $\mathcal{T}$ in order to express implicit of explicit malicious behavior.*

We define the *trigger signal dimension* d of a HT represented by a set of trigger states $\mathcal{T}$ as follows:

Definition 8. ***Trigger Signal Dimension $d(\mathcal{T})$*** *of a HT is defined as $d(\mathcal{T}) = \max_{Trig \in \mathcal{T}} |Trig|$.*

In other words, the trigger signal dimension shows the width of the widest trigger signal bit-vector of a hardware trojan. Obviously, it becomes difficult to detect trojans which only have high dimensional sets of trigger states, i.e. large trigger signals. The set of possible values of a given trigger signal $Trig$ grows exponentially in $d = |Trig|$ and only one value out of this set can be

[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*.

related to the occurrence of the corresponding trigger condition. 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.

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, we define the *payload propagation delay* t as follows:

Definition 9. ***Payload Propagation Delay $t(\mathcal{T})$*** *of a hardware trojan represented by a set of trigger states $\mathcal{T}$ is defined 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 signal is asserted till 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 the occurrence of the specific counter value (i.e. trigger condition). 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 5 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 $0 \leq i \leq u-1$, 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

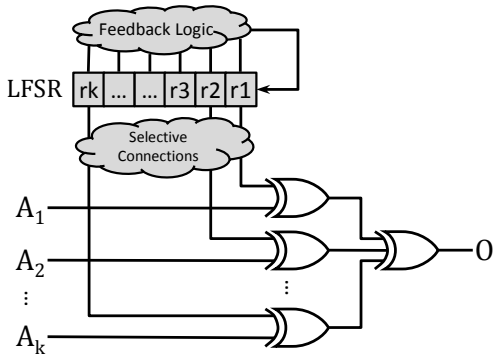


Figure 5: *k-XOR-LFSR*: A general H_D trojan.

$\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \leq i < u-j\}$, 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 $\mathbf{r}^{u-j}|P \in \{\mathbf{r}^i|P : 0 \leq i < u-j\}$ is at least equal to the probability that $\{\mathbf{r}^i|P : 0 \leq i < u-j\} = \{0,1\}^d$, which is (by the union bound)

$$\geq 1 - \sum_{w \in \{0,1\}^d} Prob(\{\mathbf{r}^i|P : 0 \leq i < u-j\} \subseteq \{0,1\}^d \setminus \{w\})$$

$$= 1 - \sum_{w \in \{0,1\}^d} (1 - 1/2^d)^{u-j} \approx 1 - 2^d e^{-(u-j)/2^d}$$

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 $d \leq \log(u-j) - \log(\log(u-j)\log k + \log\log k)$, 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 on $d(\{Trig\})$, i.e.,

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

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

As explained in section 2.2, due to the implicit malicious behavior it may not always be possible to distinguish a malicious output from a normal output just by monitoring the output ports. Consequently, the implicit malicious behavior adds to the stealthiness of the HT since it creates a possibility of having a false negative under logic testing based techniques. We quantify this possibility by defining the *implicit behavior factor* α as follows:

Definition 10. ***Implicit Behavior Factor $\alpha(\mathcal{T})$*** *of a HT represented by the set of trigger states $\mathcal{T}$ is defined as $\alpha(\mathcal{T}) = \max_{Trig \in \mathcal{T}} \alpha(Trig)$ where $\alpha(Trig)$ shows the probability that, given the trigger state $Trig$ occurs, it will lead to* implicit *malicious behavior.*

We notice that the higher the value of α, the lower the chance of detection by logic testing even if the HT gets triggered and hence the higher the overall stealthiness of the HT. (For our *k-XOR-LFSR* trojan, clearly $\alpha(Trig) = 0$ since it *always* produces incorrect output once the HT gets triggered.)

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 $\geq \log k - 2 \log \log k$.

The choice of parameters t and d significantly affects α of the HT. α as a function of t and d is decreasing in both t and d. Reducing t means that explicit malicious behavior may not have had the chance to occur, hence, the probability α that no explicit malicious behavior is seen increases. Similarly, reducing d can increase α since as a result of smaller d, there may not exist a set of trigger signals $\mathcal{T}$ that represents the HT 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 HT.

We notice that the individual wires of a HT trigger signal of dimension d are logically/physically located in the close vicinity of each other in the circuit netlist/layout. This is because eventually these wires need to coordinate (through some combinational logic) with each other to perform the malicious operation. Based on this observation, we introduce the idea of *locality* in gate level circuits, similar to the region based approach in [8].

Consider the simple combinational circuit from Figure 6a. Based on this circuit, we draw a *locality graph* shown in Figure 6b 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. For any two nodes (i.e. wires) i and j in a locality graph, we define $dist(i, j)$ as the shortest distance between i and j. In other words, $dist(i, j)$ represents the minimum number of basic combinational or sequential logic levels (e.g. logic gates and/or flipflops) between wires i and j. E.g. $dist(E, B) = dist(E, C) = 1$, whereas $dist(E, O) = dist(E, A) = 2$.

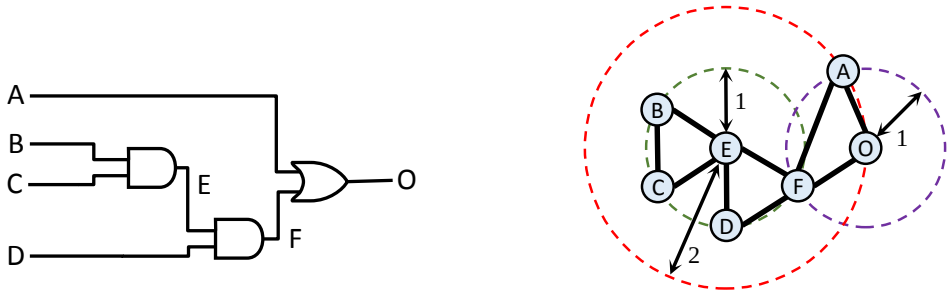
Definition 11. ***Locality*** $l(\mathcal{T})$ *of a HT represented by the set of trigger states* $\mathcal{T}$ *is defined as:*

$$l(\mathcal{T}) = \max_{Trig \in \mathcal{T}} \left(\max_{0 \leq i,j < |Trig|} dist(Trig[i], Trig[j]) \right)$$

A low value of $l(\mathcal{T})$ shows that the trigger signal wires of the HT are in the close vicinity of each other and vice versa. Having a notion of locality significantly reduces the computational complexity of HaTCh (cf. section 5.1.2).

4 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.



(a) A simple circuit

(b) Locality Graph of the circuit

Figure 6: Example of Locality Graph

4.1 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.

4.1.1 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.

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

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

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.

4.2 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 $OutSpec$.

4.2.1 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 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.

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

4.2.2 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}.

4.2.3 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.

5 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:

$Core$:	The IP core under test.
$\mathcal{U}$:	A user distribution with a pt sampling algorithm SAMPLE where each $User \leftarrow$SAMPLE$(\mathcal{U})$ is a pt-algorithm.
t, d, α:	Parameters characterizing the hardware trojan.
λ:	A security parameter.
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.

5.1 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 12. ***t-Legitimate States:*** *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.)*

Definition 13. ***Projections:*** *We define a vector* $\boldsymbol{z}$ *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*

Algorithm 2 HaTCh Algorithm

1: **procedure** HaTCh($Core, \mathcal{U}, t, d, \alpha, \lambda, \rho$)
2: $k = \lceil \frac{\lambda}{log_2(1/\alpha)} \rceil$, $B = \emptyset$
3: **for all** $1 \leq i \leq k$ **do**
4: $B_i \leftarrow$ Learn($Core, \mathcal{U}, t, d, \rho$)
5: **if** $B_i =$ "Trojan-Detected" **then**
6: **return** "Trojan-Detected"
7: **else**
8: $B = B \cup B_i$
9: **end if**
10: **end for**
11: $Core_{Protected} =$ Tag($Core, B$)
12: **return** $Core_{Protected}$
13: **end procedure**

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 14. ***$H_{t,\alpha,d}$:*** *$Core \in H_{t,\alpha,d}$ if and only if it is represented by a set of trigger states $\mathcal{T}$ with $t(\mathcal{T}) \leq t$ and $d(\mathcal{T}) \leq d$ 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:*

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

We refer to the minimum α that satisfies $\mathcal{C}2$ for $\mathcal{T}$ as the implicit behavior factor $\alpha(\mathcal{T})$ of the hardware trojan (cf. Definition 10).

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

[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** $1 \leq i \leq |States| - t$ **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 B
19: **return** B ▷ The Blacklist
20: **end if**
21: **end procedure**

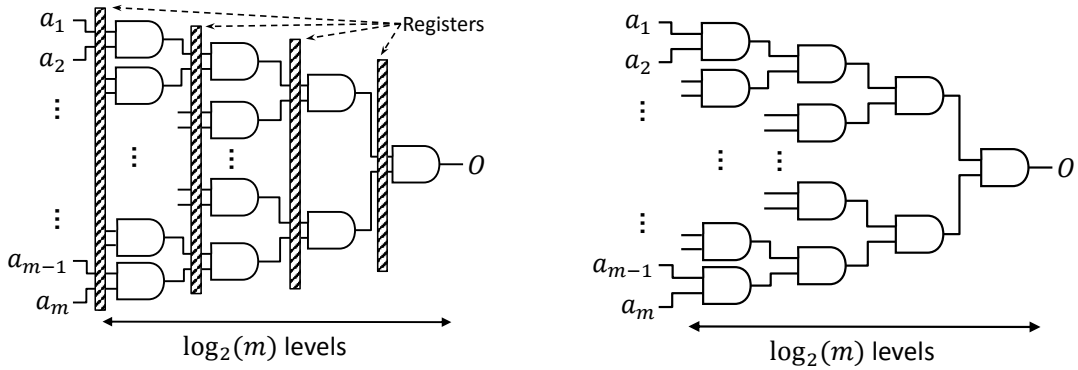
process is repeated until the blacklist size does not reduce any further, i.e. until $1/\rho$ consecutive test patterns do not reduce the blacklist anymore. This means that neither a false nor a true positive would have been generated if this blacklist were used for the tagging phase. For this reason 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 blacklist B generated by algorithm 3 is equal to

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

for the sampled $User$ (line 5 in algorithm 3). B may contain two types of wires; 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.

5.1.1 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



(a) Seen by VeriTrust or FANCI

(b) Seen by VeriTrustX or FANCIX

Figure 7: An m-input AND gate implemented by 2-input AND gates.

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 $1 - 2^{-\lambda}$.

5.1.2 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, 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 exploit the *locality* in gate level circuits (cf. Definition 11). 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:

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

5.1.3 Complexity: HaTCh vs. Existing Techniques

In this subsection, we compare the computational complexities of existing state of the art trojan detection schemes, namely VeriTrustX and FANCIX, with the complexity of HaTCh. Consider a simple m-input pipelined AND gate as shown in Figure 7a. It is implemented by a tree of 2-input AND gates and registers, where the tree has $\log_2(m)$ levels (i.e. sequential stages). Since without any knowledge of the HT, both VeriTrustX and FANCIX will need to monitor all the sequential

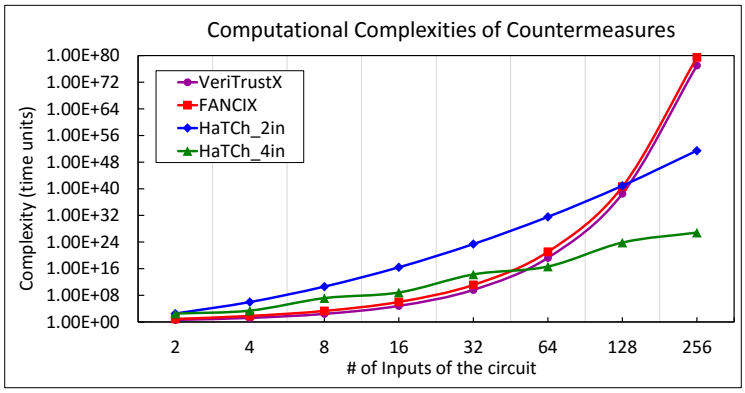


Figure 8: Comparison of computational complexities of different Hardware Trojan countermeasures.

stages of the design, therefore it will be considered a purely combinational logic block as shown in Figure 7b.

As explained earlier, in order to avoid any false negatives caused by the implicit malicious behavior, VeriTrustX must monitor the activation history of each entry in the truth table instead of the terms in SOP/POS form of the boolean function. This leads to an exponential computational complexity i.e. $O(2^m)$ as the truth table has 2^m entries in total.

Similarly FANCIX also needs to go through each entry of the truth table to compute the control value (CV) of a single input, and this process is repeated for each of the m inputs. This leads to a computational complexity $O(m2^m)$. We do notice that the basic version of FANCI suggests to reduce the complexity by randomly selecting a reasonable number of entries in the large truth table to compute the control values. This method can also be used in FANCIX to reduce the complexity. However, this optimization can potentially lead to a higher probability of false negatives. Therefore, for maximum security, we consider a full scaled version of FANCIX which is a fair comparison with HaTCh.

HaTCh, on the other hand, has $O((2n^2)^d)$ complexity (i.e. a single exhaustive iteration which is sufficient in this case) which scales much better with the size of circuit. Here n represents the total number of wires in the circuit and d represents the worst case trigger signal dimension. For the circuit in Figure 7b (i.e. 2-input AND gate implementation) $n = 2m - 1$, whereas $d = \log_2(m) + 1$ which shows the maximum number of levels over which a trigger signal can be spread. Notice that if 4-input AND gates are used to implement this circuit (which is typically common in modern synthesis tools and ASIC libraries) then n and d reduce to $n = m + \lceil \frac{m-1}{3} \rceil$ and $d = \lceil \log_4(m) \rceil + 1$ respectively, leading to even lower computational complexity.

Figure 8 shows a comparison of (runtime) complexities of different countermeasures. $HaTCh\_2in$ and $HaTCh\_4in$ show the complexities for the m-input AND gate circuit implemented with 2-input and 4-input AND gates respectively. Notice that for $n = 2m - 1$ and $d = \log_2(m) + 1$, the complexity of HaTCh is $O(m^4 4^{(\log_2 m)^2})$ which is sub-exponential in m and not exponential in m as for FANCIX and VeriTrustX. Clearly HaTCh is the best choice for practical circuits, such as encryption engines which have 128 or more inputs, as it scales better with the number of inputs (i.e. circuit size) whereas the complexities of both VeriTrustX and FANCIX shoot up rapidly.

5.2 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 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.

6 Evaluation

In this section, we evaluate our HaTCh tool for Trusthub [5] benchmarks, trojan designs proposed by DeTrust which defeat VeriTrust and FANCI, and also for our newly designed k-XOR-LFSR trojan. 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 in HaTCh to minimize the area overhead. Finally we present and discuss the experimental results.

6.1 Characterizing Trusthub Benchmarks

Table 1 shows the relevant[7] benchmarks from Trusthub categorized according to the HaTCh framework. *St-D* group (i.e. H_D trojans) 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[8] are listed in Table 1. 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 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

[7] Not all of the benchmarks from TrustHub are listed in Table 1, because some of them have no payload, such as RS232-T200. Similarly the payloads of some other benchmarks are harmless which would be removed by synthesis tools. E.g. RS232-T1800, which just adds three inverters to waste energy.

[8] 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}

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. In our model, some TrustHub benchmarks, e.g. MC8051 series, are considered to be in H_{ND} group because of their flexible design specifications. The specification of a processor is relatively flexible about the timing/ordering of outputs (e.g. instructions execution) due to some unpredictable factors like interrupt requests etc. This flexibility, however, makes it harder for HaTCh to verify the functional correctness of the design. However, if there exists a precise and strict model for these cores, HaTCh is still able to analyze them. Notice that trigger activated *Si* trojans can actually be detected by HaTCh provided that these trojans do not get triggered during the HaTCh learning phase (so that their trigger related wires still remain blacklisted).

6.2 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.

6.3 Experimental Results

6.3.1 TrustHub s-Series Benchmarks

Figure 9 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

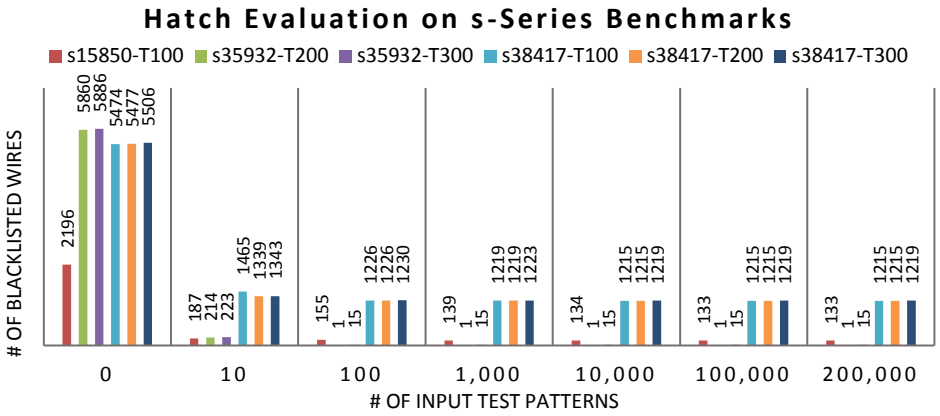


Figure 9: Blacklist size of s-Series with $d = 1$

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%

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.

6.3.2 TrustHub RS232 Benchmarks

We first test five RS232 benchmarks (namely RS232-T{100, 300, 500, 600, 700}) which have dimension $d = 1$. The blacklists produced by HaTCh for these benchmarks only contains a single wire, i.e. the trigger signal. Hence, these benchmarks do not incur any additional area overhead.

We also test some RS232 benchmarks (i.e. RS232-T{300, 1200, 1300}) using parameters $d = 2$ and locality $l = 1$ just to get a real estimate of HaTCh overheads for higher dimensions, even though these benchmarks belong to $d = 1$ HT subclass. Table 3 shows the area overheads of these benchmarks. We see that even with $d = 2$, 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.

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%

6.3.3 New Trojans by DeTrust

As TrustHub benchmarks are quite simple and also outdated, therefore in order to demonstrate the strength of HaTCh tool, we implement the trojans proposed by DeTrust [23] (which defeat VeriTrust and FANCI) and test them with our HaTCh tool.

After synthesis on a standard ASIC library, the FANCI-defeating trojan from Figure 3b results in a trojan with trigger dimension $d = 1$. Hence, it is very straightforward to detect this trojan using the parameter $d = 1$ for HaTCh. In our experiment, after a learning phase of only $100,000$ random unique input patterns, the blacklist produced by HaTCh contains only one wire which is the trigger signal of this trojan.

The trojan defeating VeriTrust as depicted in 2b can be detected by HaTCh using the parameter $d = 2$ under the assumption that the implicit malicious behavior does not occur during the learning phase. Whereas without this assumption, HaTCh is still able to detect this trojan with $d = 3$ even if the implicit malicious behavior occurs.

6.3.4 k-XOR-LFSR Trojan

HaTCh can also detect our newly designed *k-XOR-LFSR* trojan. As an example, we implemented the *k-XOR-LFSR* trojan for $k = 4$ which leads to a trigger dimension $d = 2$. Therefore, HaTCh requires the parameter $d = 2$ in order to detect it. The corresponding area overhead for this trojan in our experiment is negligible as for now we tested it as a standalone circuit. However, depending upon the IP core in which this trojan is embedded, the area overhead will vary in proportion to the IP core size. Notice that if frequency dividers are used to carefully slow down the clock frequency driving the *k-XOR-LFSR* trojan, even a small k value can take several cycles before the trojan is actually triggered.

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 analyze what kind of additional properties must be satisfied by such 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.

We also show that some existing countermeasures have serious security loopholes due to some of their fundamentally invalid assumptions. Additionally, we compare and show that, for an m input circuit, the computational complexity of HaTCh is sub-exponential in m as opposed to exponential complexity (in m) of state of the art techniques such as VeriTrust and FANCI. This comparison holds for an m-input pipelined AND gate.

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 & 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.

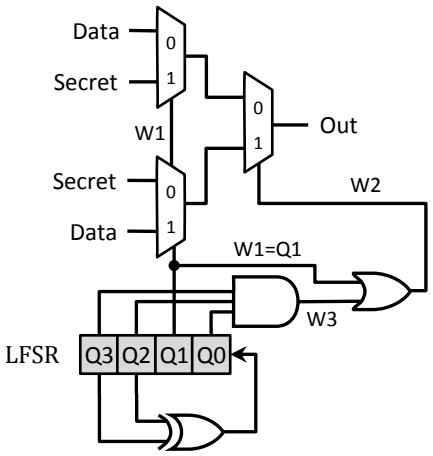
[26] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 396–410.

[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 10: A Counter-Based $H_{0,2}$ Trojan to enable the secret leakage

A Analysis of k-*XOR-LFSR*

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

The example trojan shown in Figure 10a 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 10b 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.

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

As it is clear from Figure 10b 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 still 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.