Advancing the State-of-the-Art in Hardware Trojans 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

June 16, 2016

Over the past decade, Hardware Trojans (HTs) research community has made significant progress towards developing effective countermeasures for various types of HTs, yet these countermeasures are shown to be circumvented by sophisticated HTs designed subsequently. Therefore, instead of guaranteeing a certain (low) false negative rate for a small *constant* set of publicly known HTs, a rigorous security framework of HTs should provide an effective algorithm to detect any HT from an *exponentially large* class (exponential in number of wires in IP core) of HTs with negligible false negative rate.

In this work, we present HaTCh, the first rigorous algorithm of HT detection within the paradigm of pre-silicon logic testing based tools. HaTCh detects any HT from H_D, a huge class of deterministic HTs which is orders of magnitude larger than the small subclass (e.g. TrustHub) considered in the current literature. We prove that HaTCh offers negligible false negative rate and controllable false positive rate for the class H_D. Given certain global characteristics regarding the stealthiness of the HT within H_D, the computational complexity of HaTCh for practical HTs scales polynomially with the number of wires in the IP core. We implement and test HaTCh on TrustHub and other sophisticated HTs.

Contents

1 Introduction

System on Chip (SoC) designers frequently use third party IP cores as black boxes instead of building these logic blocks from scratch, in order to save the valuable time and other resources. However, these third party IP cores can contain Hardware Trojans (HTs) which could potentially harm the normal functionality of the SoC (i.e. denial of service attack) or cause privacy leakage [2–5]. These Trojans must be detected in pre-silicon phase, otherwise an adversary can infect millions of ICs through a Trojan affected IP core.

The IP core vendors generally do not offer the RTL level source code of the IP cores because of the proprietary reasons. Therefore, digital IP cores are typically offered in the form of a netlist of common digital logic gates and memory elements. Such IP cores are termed as 'closed source' IP cores. Even if the RTL source code of the IP core is available, it might not be feasible in case of large IP cores to manually inspect the source code for possible HTs. Hence, closed source IP cores impose totally new challenges of identifying potentially malicious logic gate(s) from a netlist of hundreds of thousands of gates, without any knowledge of the RTL source code. This is analogous to finding a needle in a haystack.

Although Hardware Trojans research community has put in tremendous efforts to develop effective countermeasures to detect several kinds of HTs, however still, the HT design research has always been one step ahead of the HT detection research. State of the art HT detection schemes namely Unused Circuit Identification (UCI) [6], VeriTrust [7], and FANCI [8] have shown that they can detect all HTs from the TrustHub [9] benchmark suite. On the other hand, researchers have also shown that these schemes can still be circumvented by carefully designing new HTs [10], [11]. A recent work, DeTrust [12], in fact introduces a systematic methodology of designing Trojans that can evade both VeriTrust and FANCI.

We notice that the fundamental reason behind the possibility of circumventing these HT detection techniques is the lack of rigorous analysis about the coverage of these techniques at their design time. Typically, HT detection techniques offer guaranteed coverage for a small *constant* set of *publicly known* HT benchmarks such as TrustHub. However, later on, one may design a HT which is slightly different than previously known HTs such that it can bypass the detection schemes. Therefore, clearly the HT detection techniques must target and provide security guarantees against certain behavioral traits resulting in a larger *class* of HTs instead of targeting specific known HTs. We limit our discussions in this paper about digital IP cores and trigger activated digital HTs which have digital payloads. Also, since we want to detect any HTs inside the SoC before it gets fabricated, we restrict ourselves to the tools and algorithms which can detect HTs in pre-silicon phase. Hardware Trojans that are always active and/or exploit side channels for their payloads [13] [14] [15] are out of scope of this paper.

Recent work [1] discovers several advanced properties (i.e. d, t, α, l) of trigger based deterministic HTs resulting in a large group called H_D. These properties characterize the stealthiness of the HTs, and hence introduce the design principles that an adversary might follow to design sophisticated HTs in order to bypass the current HT countermeasures.

In this paper, we present a powerful HT detection algorithm called ***Ha**rdware **T**rojan **C**atc**h**er* (HaTCh) that offers complete coverage and transparent security guarantees for the above mentioned large and complex class H_D of deterministic HTs. Figure 1 compares the coverages offered by different HT detection schemes for the HT class H_D. Although VeriTrust and FANCI can detect publicly known TrustHub HTs, it is unclear what security guarantees they offer outside TrustHub. HaTCh, on the other hand, is capable of detecting any HT from H_D, whether the HT is publicly

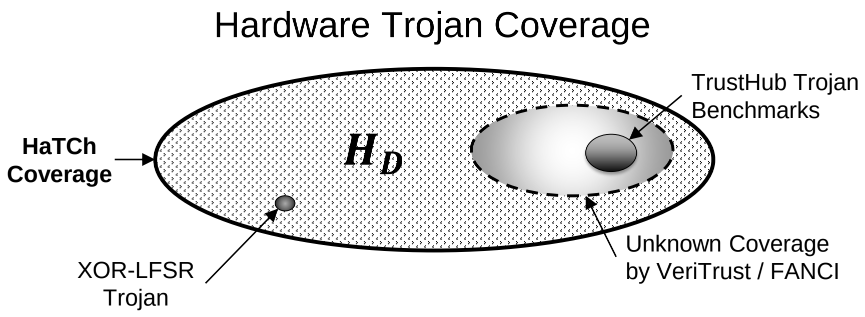

Hardware Trojan Coverage
TrustHub Trojan Benchmarks
HaTCh Coverage
H_D
XOR-LFSR Trojan
Unknown Coverage by VeriTrust / FANCI

Figure 1: Class of Deterministic Hardware Trojans H_D, and detection coverages of existing Hardware Trojan countermeasures.

known or unknown. We prove that HaTCh offers negligible (in $2^{-\lambda}$) false negative rate and a controllable false positive rate ρ.

Our proposed HT detection tool 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. If these untrusted transitions – that are potentially related to a HT trigger circuit – ever occur in the lifetime of the circuit, an exception is raised that shows the activation of a HT. We have implemented HaTCh and our experimental as well as theoretical results demonstrate that:

- HaTCh detects all H_D trojans in $poly(n)$ time where n is number of circuit wires and the degree of the polynomial corresponds to the stealthiness of the HT embedded in the circuit. In particular, all H_D trojans from TrustHub are detected in linear time.
- HaTCh incurs low area overhead (on average 4.18% for a set of tested HT benchmarks).
- In cases where state of the art HT detection tools have a worst case exponential complexity in the number of circuit inputs, HaTCh offers orders of magnitude lower complexity.

The rest of this paper is organized as follows. Section 2 provides necessary background of HTs, briefly explains the HT class H_D w.r.t. its advanced properties followed by our threat model. Sections 3 and 4 present the basic algorithm and the detailed methodology of HaTCh, respectively. We rigorously analyze the security guarantees of HaTCh in section 5 and prove bounds on false negatives and false positives rates. In section 6 we compare HaTCh with state of the art HT detection schemes. Section 7 shows the experimental evaluation and we finally conclude in section 8.

2 Background & Threat Model

We first provide the necessary background and define various terminologies upon which we build our further discussions in this paper. In particular, we briefly review the characteristics of the HT class H_D presented in [1] which play a crucial role in the development of our detection algorithm called HaTCh .

2.1 Basic Terminologies

Some important terminologies used in [1] are as follows:

Definition 1. *A* ***Hardware Trojan*** *(HT) can be defined as malicious redundant circuitry embedded inside a larger circuit, which results in data leakage or harm to the normal functionality of the circuit once activated.*

A *trigger activated* HT activates upon some special event, whereas an *always active* HT remains active all the time to deliver the intended payload. Once activated, a HT can deliver its payload either through standard I/O channels and/or through the side channels.

Definition 2. ***Trigger condition*** *is 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*** *is a collection of physical wire(s) which the HT trigger circuitry asserts in order to activate the payload circuitry once a trigger condition occurs.*

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

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

Definition 6. ***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.*

Definition 7. ***False Negative*** *is 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.*

Definition 8. ***False Positive*** *is a scenario when a HT detection tool identifies a Trojan-free circuit to be a Hardware Trojan-infected circuit.*

2.2 Properties of H_D Hardware Trojans

A Hardware Trojan detection scheme without a well defined scope on the landscape of HTs fails to provide concrete security guarantees. For this reason, we first define the scope of our detection algorithm by restricting it to the class of trigger based deterministic Hardware Trojans, namely the class H_D introduced in [1]. H_D represents the HTs which are embedded in a digital IP core whose output is a function of only its input, and the algorithmic specification of the IP core can exactly predict the IP core behavior. Four crucial properties (d, t, α, l) of this class are also introduced in [1] which determine the stealthiness of a H_D HT having a set of trigger states $\mathcal{T}$. A brief highlight of these properties is as follows.

Trigger Signal Dimension $d(\mathcal{T})$ represents the number of wires used by HT trigger circuitry to activate the payload circuitry in order to exhibit malicious behavior. A large d shows a complicated trigger signal, hence it is harder to detect.

Payload Propagation Delay $t(\mathcal{T})$ is the number of cycles required to propagate malicious behavior to the output port *after* the HT is triggered. A large t means it takes a long time after triggering until the malicious behavior is seen, hence less likely to be detected during testing.

Implicit Behavior Factor $\alpha(\mathcal{T})$ represents the probability that given a HT gets triggered, it will not (explicitly) manifest malicious behavior; this behavior is termed as implicit malicious behavior. Higher probability of implicit malicious behavior means higher stealthiness during testing phase.

Trigger Signal Locality $l(\mathcal{T})$ shows the spread of trigger signal wires of the HT across the IP core. Small l shows that these wires are in the close vicinity of each other. Large l means that these wires are spread out in the circuit, and hence the HT is harder to detect.

A Hardware Trojan can be represented by multiple sets of trigger states $\mathcal{T}$, each having their own d, t, α, and l values. The collection of corresponding quadruples (d, t, α, l) is defined as the achievable region of the Hardware Trojan. This allows us to define $H_{d,t,\alpha,l}$ as follows:

Definition 9. *$H_{d,t,\alpha,l}$ is defined as all H_D type Trojans which can be represented by a set of trigger states $\mathcal{T}$ with parameters $d(\mathcal{T}) \leq d$, $t(\mathcal{T}) \leq t$, $\alpha(\mathcal{T}) \leq \alpha$ and $l(\mathcal{T}) \leq l$.*

In the remainder of this paper we develop HaTCh which takes parameters d, t, α and l as input in order to detect HTs from $H_{d,t,\alpha,l}$.[1]

2.3 Threat Model

A third party IP core is provided in the form of a synthesized netlist which obfuscates the HDL source code. The IP core vendor has embedded a trigger based HT in the IP core which delivers its payload via standard IO channels of the IP core. The HT could have been inserted directly by maliciously modifying the source code or by using some malicious tools to synthesize the RTL. This IP core is used in a larger design whose millions of chips are fabricated. The adversary wants to exploit this scalability to infect millions of fabricated chips by supplying an infected IP core. To prevent this, we test the IP core for potential HTs in *pre-silicon* phase, i.e. before integrating it 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.

3 HaTCh Algorithm

In this section, we present the basic HT detection algorithm of HaTCh which uses *whitelisting* approach to discriminate the trustworthy circuitry of an IP core from its potentially malicious parts.[2] In order to detect any HT in a $Core \in H_{d,t,\alpha,l} \subseteq H_D$, HaTCh takes the following parameters as input:

[1] $H_{d,t,\alpha,l} \subseteq H_{d,t,\alpha}$ where $H_{d,t,\alpha}$ represents the group of HTs which are not constrained by a certain value of the locality parameter l.

[2] The English word *Hatch* means an opening of restricted size allowing for passage from one area to another. Our HaTCh framework provides this functionality by allowing certain parts of the circuit to operate normally while restricting the operation of others.

$Core$:	The IP core under test.
$\mathcal{U}$:	A user distribution with a ppt sampling algorithm SAMPLE where each $User \leftarrow$SAMPLE($\mathcal{U}$) is a pt-algorithm.
d, t, α, l:	Parameters characterizing the Hardware Trojan.
λ:	False negative rate $\leq 2^{-\lambda}$.
ρ:	Maximum acceptable false positive rate.

Algorithm 1 shows the operation of HaTCh which processes $Core$ in two phases; a *Learning phase* and a *Tagging phase*.

3.1 Learning Phase

The learning phase performs k iterations of functional testing on $Core$ using input test patterns generated by k users from a user distribution $\mathcal{U}$ and learns k independent blacklists $B_1, B_2, \ldots, B_k$ of unused wire combinations. Here 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 the 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.

3.2 Tagging Phase

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 wire combinations is activated, a special flag will be raised to indicate the activation of a potential Hardware Trojan.

In particular, a new output signal called $TrojanDetected$ is added to the $Core$. A tree of logic gates is added to $Core$ such that a logic 1 is propagated to $TrojanDetected$ output whenever any wire combination from B takes a 'blacklisted' value. 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 registers may delay the detection of the HT by $O(\log_2(|B|d))$ cycles, however for average sized IP cores, HaTCh can produce a significantly small B with reasonable run time. Consequently, the detection delay because of pipeline registers is only a few cycles.

We notice that logic testing based approach for Trojan detection is generally pretty straightforward, and has already been proposed in [16]. However, the main edge of HaTCh over existing schemes is that it can detect a much larger class H_D of HTs which, to say the least, cannot be efficiently detected by the existing schemes.

Algorithm 1 HaTCh Algorithm

```
1:  procedure HaTCh($Core, \mathcal{U}, d, t, \alpha, l, \lambda, \rho$)
2:      $k = \lceil \frac{\lambda}{log_2(1/\alpha)} \rceil$, $B = \emptyset$
3:      for all $1 \le i \le k$ do
4:          $B_i \leftarrow$ Learn($Core, \mathcal{U}, d, t, l, \rho$)
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
```

4 Detailed Methodology of HaTCh

In order to formally model and define Hardware Trojans, and to present a detailed implementation of HaTCh, we first provide a relaxed model for the input and output behavior of the IP cores.

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[3] $User$ 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

[3]Any random coin flips necessary are stored as a common reference string in the algorithm itself. Note that user is a deterministic algorithm.

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

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 [17], 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.
- $(y_i, S_i) \longleftarrow Emulate[Core](x_i, S_{i-1})$.

Algorithm 2 $User$ interacts with $Emulate[Core]$ and verifies functional correctness and outputs the list of all the emulated states of M^{Core}.

```
1:  procedure SIMULATE($Core, User$)
2:      $g, Y_0, j, States = 1, \epsilon, 1, [\,]$
3:      $S_0, S'_0 = ResetStateCore, ResetStateSim$
4:      while $(X_j, U_j) \leftarrow User(Y_{j-1}, U_{j-1})$ do
5:          $(Y'_j, S'_j, \Delta_j) \leftarrow CoreSim(X_j, S'_{j-1})$
6:          $(x_k, \ldots, x_{k+n}) \leftarrow$ SEND$(X_j)$
7:          $(x_g, \ldots, x_{k-1}) = (\epsilon, \ldots, \epsilon)$
8:          $(x_{k+n+1}, \ldots, x_{k+n+\Delta_j}) = (\epsilon, \ldots, \epsilon)$
9:          for $i \leftarrow g, k+n+\Delta_j$ do                          ▷ Emulate
10:             $(y_i, S_i) \leftarrow Emulate[Core](x_i, S_{i-1})$
11:             if $y_i \neq \epsilon$ then $Y_j = Y_j || y_i$
12:             end if
13:             $Append(States, S_i)$                                  ▷ Update $States$
14:         end for
15:         $j, g = j+1,\ k+n+\Delta_j+1$
16:         if $Y'_j \neq Y_j$ then                                    ▷ Verification
17:             return ("Trojan-Detected", $\cdot$)
18:         end if
19:     end while
20:     return ("OK", $States$)                                        ▷ All emulated states
21: end procedure
```

$Emulate[Core]$ 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 [18] 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.

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 2 shows a detailed description of this process (U_i indicates the current state or working memory of $User$).

Notice that $User$ in algorithm 2 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 2, i.e., if the emulated output (by $Emulate[Core]$) correctly corresponds to the simulated output (by $CoreSim$). If it does not verify properly, then we say that the input sequence $X_1, X_2, \ldots, X_N$ violates the functional spec.

4.3 Legitimate States & Projections

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

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

Formally, a trigger state is a labelled binary vector, i.e., it is a pair $(P, \mathbf{x})$ where P denotes a projection and $\mathbf{x}$ is a binary vector; if $Core$ is in state $\mathbf{z}$ and $\mathbf{z}|P = \mathbf{x}$ then the trojan gets triggered. Now let $\mathcal{T}$ be a set of trigger states/signals which *represents* the hardware trojan, i.e., M^{Core} manifests malicious behavior if and only if it has passed through a state in $\mathcal{T}$. Let $\mathcal{T}$ have dimension d and payload propagation delay t, i.e., the trojan always manifests malicious behavior within t clock cycles after "it gets triggered" by a trigger signal in $\mathcal{T}$ (cf. section 2.2). 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". In the next subsection, we use this notion of t-legitimate states to learn a blacklist of suspicious wires in the IP core.

4.4 Learning Phase Algorithm

Algorithm 3 describes the operation of a single iteration in HaTCh learning phase[4] (lines 3-10 in Algorithm 1). 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

[4]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

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

process is repeated until the blacklist size does not reduce any further, i.e. until $1/\rho$ consecutive user interactions (and their resulting states) do not reduce the blacklist anymore. This means that neither a false nor a true positive would have been generated if this blacklist was used for the tagging phase. For this reason ρ becomes, statistically, the estimated upper bound on the false positive rate. A detailed proof of this bound is presented in section 5.2.

The blacklist B generated by algorithm 3 is equal to

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

for the sampled $User$ (line 5 in algorithm 3).

Notice that 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. The second type of wires in the blacklist would lead to false positives. However, the controlled nature of the HaTCh algorithm allows the user to control the false positives rate by trading off with the learning phase run time.

4.5 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 (1). If the set of trigger states $\mathcal{T}$ is not a subset of this union, then each of the k blacklists must exclude at least one trigger signal from $\mathcal{T}$ and therefore

$(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ for each of the corresponding k users $User$. The probability that both $W(User) =$ "OK" as well as $(L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset$ is at most α (by Bayes' rule) for a $H_{t,\alpha,d}$ trojan. We conclude that the probability that the set of trigger states $\mathcal{T}$ is not a subset of the blacklist produced by HaTCh is at most $\alpha^k \leq 2^{-\lambda}$. So, the probability that the tagging circuitry will detect all triggers from $\mathcal{T}$ is at least $1 - 2^{-\lambda}$. A detailed proof of this bound is presented in section 5.1.

4.6 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 1, 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:

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

We conclude that HaTCh runs in $poly(n)$ time where the degree of the polynomial corresponds to the maximum value of the trigger signal dimension d of the HT that can be detected in this much time. Notice that in general, the degree of this polynomial is much smaller than the total number of wires that define the HT itself.

In order to reduce the computational complexity, we exploit the *locality* in gate level circuits (cf. section 2.2). Let n_l denote 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 $n \cdot \binom{n_l}{d-1} \leq n \cdot n_l^{d-1}$ since $n_l \ll n$ Hence, the overall complexity from (2) is reduced to:

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

5 Rigorous Security Analysis of HaTCh

For medium to large sized practical circuits, it becomes almost impossible to precisely calculate the probability of implicit malicious behavior, i.e. α (cf. section 2.2). However, α can be experimentally analyzed and approximated by *experimentally observed* implicit behavior factor α'. We define α' as follows:

$Core \in H^{\text{SAMPLE}}_{d,t,\alpha'}$ 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} interacting with $User$ 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:

$$Prob\Big(W(User) = \text{"OK"} \Big| (L_t(User)|\mathcal{P}_d) \cap \mathcal{T} \neq \emptyset\Big) \leq \alpha'$$

where the probability is over $User \leftarrow \textsc{Sample}(\mathcal{U})$.

Depending upon the sampling algorithm $\textsc{Sample}$, $\alpha' \geq \alpha$ (bad for detection) or $\alpha' \leq \alpha$ (good for detection). However, we assume that $\textsc{Sample}$ algorithm samples close to the actual distribution: If the distributions $\textsc{Sample}(\mathcal{U})$ and $\mathcal{U}$ are equal and $t = 0$ in $\mathcal{C}2$, then α' becomes the actual implicit behavior factor α.

Notice that reducing t to 0 in $\mathcal{C}2$ allows a trigger signal to manifest also during the last t states produced by $\textsc{Simulate}(Core, User)$. For such a trigger signal the payload will less likely manifest in malicious behavior as it will have less than t cycles to progress to the output, hence, for such trigger signals $W(User)$ = "OK" is more likely. This means that the overall probability in $\mathcal{C}2$ (calculated as a weighted average over trigger signals which manifest during the last t cycles and trigger signals which manifest before the last t cycles) increases as t reduces to 0. So, if distribution $\textsc{Sample}(\mathcal{U})$ equals $\mathcal{U}$, then $H_{d,t,\alpha'} \subseteq H^{\textsc{Sample}}_{d,t,\alpha'}$.

The next section proves an upper bound of false negative probability for $H^{\textsc{Sample}}_{d,t,\alpha'}$. So, if $\textsc{Sample}$ algorithm samples close to the actual distribution then this result holds for $H_{d,t,\alpha'}$.

5.1 False Negatives

As discussed in the previous section, if $\textsc{Sample}$ algorithm samples close to the actual distribution, then the theorem below also holds for $H_{d,t,\alpha}$ with α interpreted as the implicit behavior factor:

Theorem 1. *For an IP core in $H^{\textsc{Sample}}_{d,t,\alpha}$ analyzed by* $\textsc{HaTCh}$*($Core, \mathcal{U}, d, t, \alpha, l, \lambda, \rho$), the probability of a false negative is upper bounded by α^k, where α is the experimentally observed implicit behavior factor of the IP core and k is the number of iterations in HaTCh learning phase.*

Proof. The final blacklist B is a union of the k blacklists $B_1, B_2, \ldots, B_k$ learned by HaTCh, i.e. $B = B_1 \cup B_2 \cup \cdots \cup B_k$. Each blacklist B_i initially includes all d-dimensional projections of wires, i.e. $B_i = \mathcal{P}_d \times \{0,1\}^d$. The projections that activate are then whitelisted from B_i one by one during the functional testing.

Let a HT be represented by a set of trigger states $\mathcal{T}$. A false negative under HaTCh is only possible if:

1. The IP core passes the functional testing during the learning phase, and
2. The tagging circuitry cannot flag a malicious behavior, i.e. $\exists_{T\in\mathcal{T}}\ T \notin B$

Since the blacklist $B = B_1 \cup B_2 \cup \cdots \cup B_k$, therefore for a false negative the trigger state $T \in \mathcal{T}$ must be whitelisted from all sub-blacklists B_i for $1 \leq i \leq k$. For $U_i \leftarrow \textsc{Sample}(\mathcal{U})$ and $(W(U_i), States_i) \leftarrow \textsc{Simulate}(Core, U_i)$, this probability is given by:

$$Prob(FN) = Prob\left(\underset{1\leq i\leq k}{\forall} W(U_i) = \text{``OK''} \wedge \underset{1\leq i\leq k}{\forall} T \notin B_i\right)$$

By definition of t-Legitimate states, the probability that $\exists_{T\in\mathcal{T}}\ T \notin B_i$ is the same as the probability that $L_t(U_i) \cap \mathcal{T} \neq \emptyset$. Therefore, $Prob(FN)$ is:

$$\begin{aligned} &= \prod_{1\leq i\leq k} Prob\Big(W(U_i) = \text{``OK''} \wedge L_t(U_i) \cap \mathcal{T} \neq \emptyset\Big) \\ &\leq \prod_{1\leq i\leq k} Prob\Big(W(U_i) = \text{``OK''} \Bigm| L_t(U_i) \cap \mathcal{T} \neq \emptyset\Big) \\ &\leq \alpha^k \end{aligned}$$

by definition of experimentally observed implicit behavior factor α.

$\square$

5.2 False Positives

Consider a user distribution $\mathcal{U}$ and a user $U \leftarrow \mathcal{U}$ which interacts with the IP core *Core*, i.e.,

$$(w, States) \leftarrow \textsc{Simulate}(Core, U).$$

In the following discussion, a subscript 's' of *Prob* represents the scenario when *Prob* is over users U sampled by SAMPLE algorithm, i.e., $U \leftarrow \textsc{Sample}(\mathcal{U})$. No subscript '$s$' represents the scenario when *Prob* is over users U drawn from $\mathcal{U}$.

In order to do a precise analysis of false positives, we make the following assumptions:
Significance Assumption: Algorithm LEARN outputs a blacklist $B = B_1 \cup B_2 \cup \cdots \cup B_k$ where (1) the complement of B_j equals the set

$$\bar{B_j} = \bigcup_{1 \le i \le L_j} States[i] \tag{3}$$

for some index L_j and (2) B_j is not decreased for another $p\Delta$ cycles after L_j (parameters p and Δ will be chosen when we describe our statistical assumptions), i.e.

$$\forall_{1 < i \le p\Delta} States[L_j + i] \notin B_j. \tag{4}$$

By $(4)_j$ and $(3)_j$ we denote properties (4) and (3) for B_j.

We assume that there is significance to the event that each iteration of LEARN produces a L_j and B_j for which it is likely that for a period of another $p\Delta$ states, no decrease in the B_j is observed. I.e., if one would look for another multiple of $p\Delta$ states, we assume that again with significant probability no decrease in B_j will be observed. In mathematical notation we assume that

$$Prob_s\left(\forall_{1 \le j \le k} (4)_j \Big| \forall_{1 \le j \le k} (3)_j\right) \ge 1 - \beta \tag{5}$$

where $1 - \beta$ represents the significance.

Statistical Assumptions:

(a) We assume that Δ is sufficiently large such that, for all i, the following relation holds:

$$\begin{aligned} & Prob\Big(States[i] = S_1 \wedge States[i+\Delta] = S_2\Big) \\ &= Prob\Big(States[i] = S_1\Big) \cdot Prob\Big(States[i+\Delta] = S_2\Big) \end{aligned}$$

In other words, Δ is large enough such that if the IP core is interacting with U then any two states in the resulting sequence of state transitions of the IP core that are Δ transitions away from each other are uncorrelated.

For practical circuits, this is a valid assumption. E.g. for a pipelined AES circuit, if the pipeline takes less than Δ cycles, then the internal states of the IP core Δ cycles apart correspond to different independent plain texts input to the AES circuit and therefore the internal states that are Δ cycles apart correspond to independent identical distributed values on the internal wires/registers (hence, independent states).

(b) We assume that sampling of the user U by Sample(U) algorithm is done close to the real distribution $\mathcal{U}$, i.e., for all i, $Prob_s(States[i] = S) \approx Prob(States[i] = S)$.

(c) Finally, we assume that $Prob(States[i] = S)$ is independent of i. Continuing the AES example above, this is a valid assumption; the states resulting from independent identical distributed plain texts being input consecutively to an AES circuit, each have the same individual distribution (even though they may be correlated if they are less than Δ transitions away from each other). For completeness, we notice that if we only assume periodic behavior ($Prob(States[i_0] = S) = Prob(States[i_1] = S)$ if i_0 equals i_1 modulo a period), then the results below generalize.

Given this assumption, we can define

$$\rho(B) = Prob(States[i] \in B)$$

for sets B.

Analysis: Let $(6)_j$ denote the property

$$\underset{1<i\leq p}{\forall} States[L_j + i\Delta] \notin B_j. \tag{6}$$

By the significance assumption,

$$1-\beta \leq Prob_s\left(\underset{1\leq j\leq k}{\forall}(4)_j \middle| \underset{1\leq j\leq k}{\forall}(3)_j\right)$$

which, since $(4)_j$ implies $(6)_j$, is at most

$$Prob_s\left(\underset{1\leq j\leq k}{\forall}(6)_j \middle| \underset{1\leq j\leq k}{\forall}(3)_j\right). \tag{7}$$

Notice that all U_j are sampled independently, therefore (7) is equal to the product

$$\prod_{j=1}^{k} Prob_s((6)_j|(3)_j).$$

We derive

$$\begin{aligned}
&Prob_s((6)_j|(3)_j)\\
&= Prob_s\left(\underset{1<i\leq p}{\forall} States[L_j + i\Delta] \notin B_j\right) \text{ (by (a))}\\
&= \textstyle\prod_i Prob_s\Big(States[L_j + i\Delta] \notin B_j\Big) \text{ (by (a))}\\
&\approx \textstyle\prod_i Prob\Big(States[L_j + i\Delta] \notin B_j\Big) \text{ (by (b))}\\
&= \textstyle\prod_i \Big(1-\rho(B_j)\Big) \text{ (by assumption (c))}\\
&= (1-\rho(B_j))^p
\end{aligned}$$

The above derivations prove

$$1-\beta \leq \Big(\prod_j (1-\rho(B_j))\Big)^p.$$

Since a product of real numbers in [0,1] is at most the product of their averages,

$$1-\beta \leq \left(\Big(\sum_j (1-\rho(B_j))/k\Big)^k\right)^p = \Big(1-\sum_j \rho(B_j)/k\Big)^{kp}.$$

After reordering terms we obtain

$$\sum_j \rho(B_j) \leq k\big(1-(1-\beta)^{1/(kp)}\big) \tag{8}$$

By using $1-(1-x)^y \leq y(x+(1/(1-x)-1)^2/2)$ for $0 \leq x \leq 1$ and $0 \leq y \leq 1$, the right hand side of (8) is at most

$$k(\beta+(\beta/(1-\beta))^2/2)/(kp) = (\beta+(\beta/(1-\beta))^2/2)/p$$

and we conclude

$$\sum_j \rho(B_j) \leq (\beta+(\beta/(1-\beta))^2/2)/p. \tag{9}$$

A state leads to a false positive if it is in the blacklist B (which means that it is being flagged) while it is not one of the trigger states of a set of trigger states $\mathcal{T}$ representing a potential hardware Trojan. By using (9) we get

$$\begin{aligned}
&Prob(State \text{ leads to a FP})\\
&\leq Prob(State \in B \setminus \mathcal{T})\\
&\leq Prob(State \in B)\\
&= Prob_s(State \in B) \text{ (by assumption (b))}\\
&\leq \textstyle\sum_j Prob_s(State \in B_j) \text{ (by the union bound)}\\
&= \textstyle\sum_j \rho(B_j)\\
&\leq (\beta+(\beta/(1-\beta))^2/2)/p
\end{aligned}$$

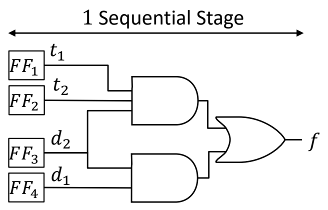
By choosing $p = 1/\rho$ and significance $1-\beta = 1/2$ we get:

Theorem 2. *For statistical assumptions (a), (b), and (c), and significance* 1/2*,* HaTCh*($Core, \mathcal{U}, d, t, \alpha, l, \lambda, \rho$) leads to a tagging circuitry which flags non Hardware Trojan trigger events (i.e., false positives) with probability at most ρ.*

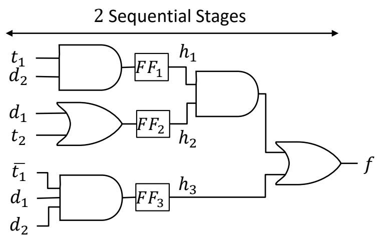
We notice that the analysis leading to the above theorem states that the more likely our observations in HaTCh, i.e., significance $1-\beta$ is closer to 1, then the more we believe a tighter upper bound $\beta(1+(\beta/(1-\beta)^2)/2)\rho \approx \beta\rho$ on the probability of false positives.

6 Comparison with Existing Techniques

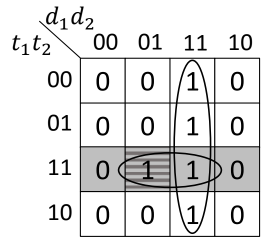
State of the art HT detection schemes are typically based on *trust verification* with the intuition that a HT almost always remains inactive in the circuit to pass the functional verification. *Unused Circuit Identification* (UCI) [6] is one of the first such techniques which distinguishes minimally used logic from the other parts of the circuit. However, due to functional verification constraints, the whole designs cannot be activated and analyzed in optimal time, and hence the scheme identifies



(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 $(t_1, t_2) = (1, 1)$.

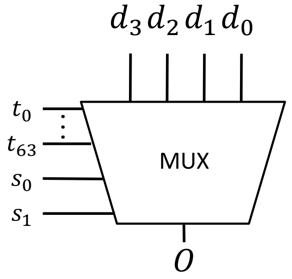
large portions of the design to be 'unused' and consider them as potential HTs. This results in a high false positive rate, and recent works have even succeeded in defeating this scheme [10], [11]. Later techniques have improved over UCI both in terms of security and performance.

VeriTrust: *Veritrust* [7] 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.

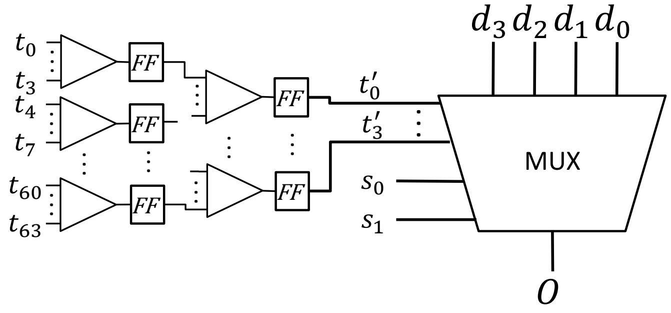
FANCI: Waksman *et al.* presents FANCI [8] which applies boolean function analysis to flag suspicious wires in a design which have weak input-to-output dependency. A *control value* (CV), which represents the percentage impact of changing an input on the output, is computed for each input in the combinational 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* [12] 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



(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.

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 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.

6.1 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*.

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. 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[5].

6.2 Computational Complexity Comparison

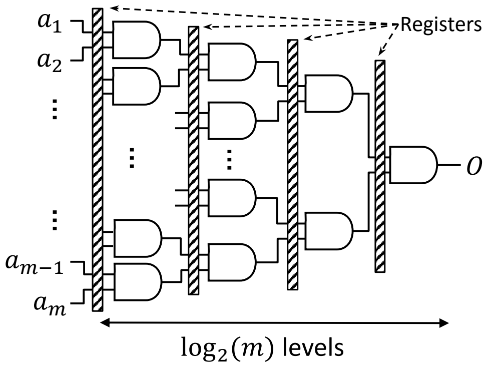
In this subsection, we compare the computational complexities of existing state of the art trojan detection schemes, namely VeriTrustX and FANCIX, with the complexity of HaTCh.

In order to analyze the worst case complexities of different HT detection techniques, we consider a simple circuit shown in Figure 4a as our *"test subject"*, i.e., as if this is the HT circuit that needs to be detected[6]. This circuit implements a m-input pipelined AND gate using 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 stages of the design, therefore it will be considered a purely combinational logic block as shown in Figure 4b.

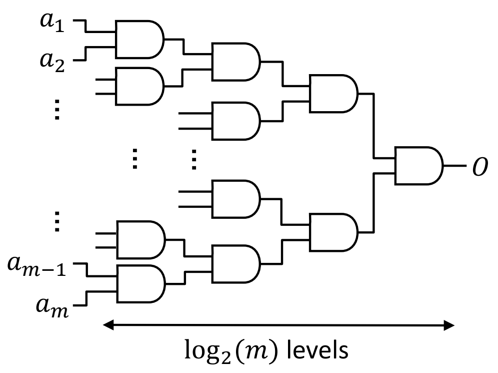
As explained earlier, in order to avoid any false negatives caused by the implicit malicious behavior, VeriTrustX must monitor the activation history of each entry in the truth table instead of the terms in SOP/POS form of the boolean function. This leads to an exponential computational complexity i.e.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

[5]Since we add tagging circuitry, therefore even if we don't catch the HT during learning phase, and if the final circuit expresses malicious behavior, the tagging circuitry detects it with overwhelming probability.

[6]Notice that, for real, this circuit is not meant to implement a hardware trojan functionality. The sole purpose of selecting this simple circuit is to conduct the worst case complexity analysis.



(a) Original implementation.



(b) Seen by VeriTrustX or FANCIX.

Figure 4: An m-input AND gate implemented by 2-input AND gates.

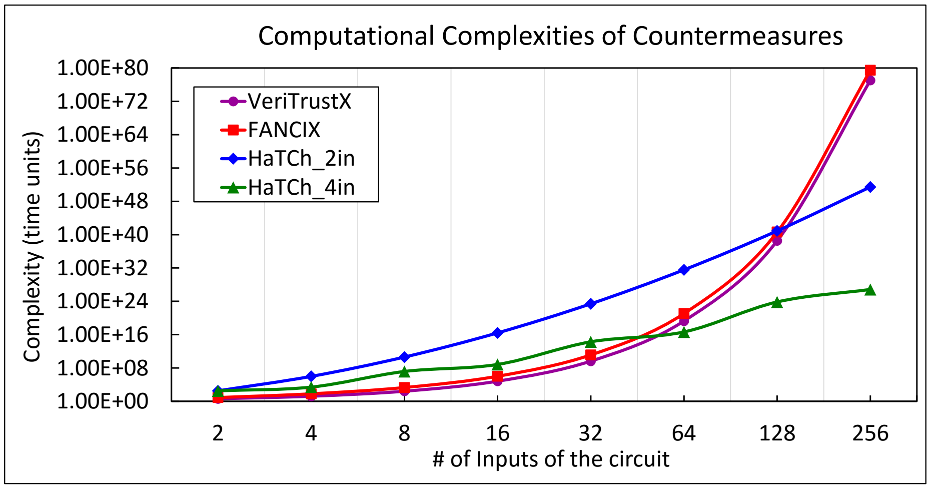


Figure 5: Comparison of computational complexities of different techniques.

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 $\Omega(m2^m)$. 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.

Since there are only $\log_2(m)$ sequential stages over which this type of HT can be spread, HaTCh can detect this HT by using $d = \log_2(m)$ in the worst case. Therefore, if n represents the total number of wires in the circuit then HaTCh needs to monitor a search space of n^d wire combinations (i.e. projections). Hence HaTCh can detect the HT with a complexity $O(n^{\log_2(m)})$ or $O(2^{d\log_2(n)})$ as compared to $\Omega(2^m) = O(2^{2^d})$ offered by existing techniques. In other words, the worst case complexity of HaTCh is orders of magnitude lower than that of VeriTrustX and FANCIX.

This exponential vs. double exponential behavior (in d) is depicted in Figure 5 which shows computational complexities of different countermeasures against the number of inputs $m = 2^d$ of the circuit from Figure 4. Notice that both horizontal and vertical axes in this plot show a logarithmic scale. For this particular circuit, n is given by $n = 2m - 1$, i.e. the sum of all internal nodes (wires) and the leafs of the tree. As it can be seen, HaTCh offers orders of magnitude lower complexity, particularly for circuits with number of inputs $m > 32$. Therefore for practical circuits, such as encryption engines which have 128 or more inputs, HaTCh is more feasible as it scales better with the number of inputs (i.e. circuit size) whereas the complexities of both VeriTrustX and FANCIX shoot up rapidly.

Although in practice, the HT is a part of a larger circuitry, we notice that the complexities of HaTCh and the other techniques scale in the same fashion since the additional wires of the larger circuit increase the search space for all these detection techniques.

7 Evaluation

In this section, we evaluate our HaTCh tool for Trusthub [9] benchmarks, Trojan designs proposed by DeTrust which defeat VeriTrust and FANCI, and also for a 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.

7.1 Characterization of TrustHub Benchmarks

The definitional framework introduced by [1] provides a concrete characterization of the TrustHub [9] Hardware Trojans benchmark suite. We revisit that analysis here briefly, which helps us to pick concrete parameter settings for HaTCh in order to detect these HTs. Since HaTCh is a tool to be used in pre-silicon phase, we are only interested in the class of deterministic HTs which use standard I/O channels to deliver their payloads, i.e. the class H_D as per the terminology used in [1].

Table 1 presents this class with subclassifications based on the specific properties of the HTs, namely d, t, and α. In our experiments, we set the input parameters to HaTCh according to the values listed in this table. All these Trojans happen to have the simplest trigger signal having a dimension $d = 1$, whereas the t and α values are observed estimated values. The methodology adapted by [1] to estimate t and α values is summarized as follows. To determine t values, simply the minimum number of registers between the trigger signal wires(s) and the output port of the IP core is computed. In order to estimate α values, first 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) is found. Then for each individual logic gate, the probability of propagating a logic 1 (considering that the trigger wire(s) get a logic 1 upon a trigger event) is computed. Finally an aggregate probability of propagation is computed by multiplying all the probabilities of each logic gate in the chain, which gives the value $1 - \alpha$. This estimated α relates to the *experimentally observed* implicit behavior factor α' in our security analysis (cf. section 5).

Table 1: Classification of Trusthub Benchmarks w.r.t. the definitional framework of [1]

	d	t	α	***Benchmarks***
Hardware Trojans Class H_D	1	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

7.2 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 is a side channel based HT, 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 [19] to synthesize the RTL design. Next, we perform post-synthesis simulations with self checking testbenches using Mentor Graphic's ModelSim [18] 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

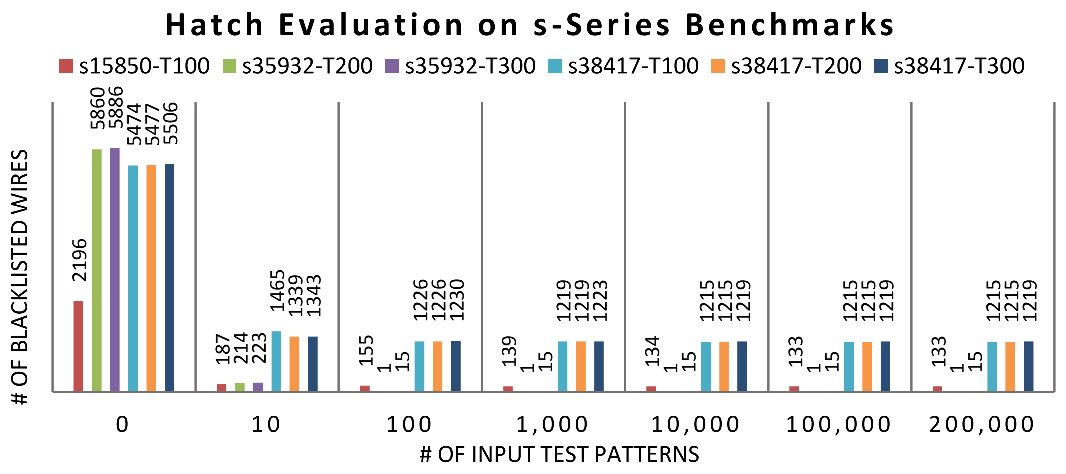


Figure 6: Blacklist size of s-Series with $d = 1$

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.

7.3 Experimental Results

7.3.1 TrustHub s-Series Benchmarks

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

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

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

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

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

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

7.3.2 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.

7.3.3 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 [12] (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

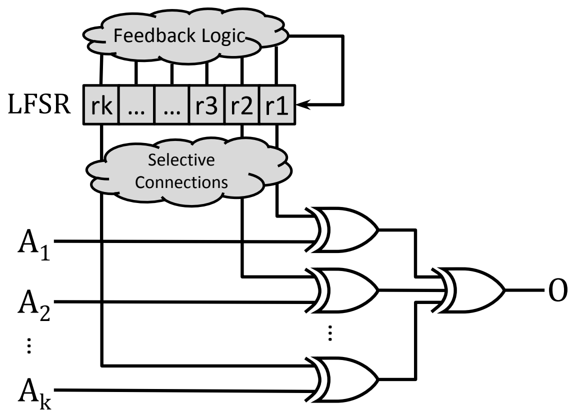


Figure 7: *k-XOR-LFSR* Hardware Trojan [1].

phase. Whereas without this assumption, HaTCh is still able to detect this trojan with $d = 3$ even if the implicit malicious behavior occurs.

7.3.4 k-XOR-LFSR Trojan [1]

HaTCh can also detect the H_D Trojan, shown in Figure 7, designed by [1] which is called *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 HT in our experiment is negligible as for now we tested it as a standalone circuit. However, depending upon the IP core in which this HT 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* HT, even a small k value can take several cycles before the HT is actually triggered.

8 Conclusion

We provide a powerful Hardware Trojan detection algorithm called HaTCh which detects *any* Hardware Trojan from H_D, a large and complex the class of "deterministic Hardware Trojans". HaTCh detects all H_D Trojans with a time complexity that is polynomial in number of circuit wires, where the degree of the polynomial corresponds to the stealthiness of the Hardware Trojan embedded in the circuit. In particular, all H_D Trojans from the TrustHub benchmark suite are detected in linear time. We conclude that our work opens up new avenues for the Hardware Trojans research community to come up with other and possibly more efficient algorithms and methodologies to detect complex Trojans that are still publicly unknown.

Acknowledgments

This project was supported in part by AFOSR MURI under award number FA9550-14-1-0351.

References

[1] S. K. Haider, C. Jin, and M. van Dijk, “Advancing the state-of-the-art in hardware trojans design,” arXiv:1605.08413, 2016, http://arxiv.org/abs/1605.08413.

[2] 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.

[3] 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.

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

[5] 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*. IEEE Press, 2013, pp. 399–404.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.*

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] “ModelSim, Mentor Graphics Inc.” www.mentor.com , http://www.model.com, Wilsonville, OR.

[19] “Synopsys Inc.” http://www.synopsys.com, Mountain View, CA.