VeriTrust: Verification for Hardware Trust

Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu

CUhk REliable Computing Laboratory (CURE)
Department of Computer Science & Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
Email: {jzhang, fyuan, lxwei, zlsun, qxu}@cse.cuhk.edu.hk

ABSTRACT

Hardware Trojans (HTs) implemented by adversaries serve as backdoors to subvert or augment the normal operation of infected devices, which may lead to functionality changes, sensitive information leakages, or Denial of Service attacks. To tackle such threats, this paper proposes a novel verification technique for hardware trust, namely VeriTrust*, which facilitates to detect HTs inserted at design stage. Based on the observation that HTs are usually activated by dedicated trigger inputs that are not sensitized with verification test cases,* VeriTrust *automatically identifies such potential HT trigger inputs by examining verification corners. The key difference between* VeriTrust *and existing HT detection techniques is that* VeriTrust *is insensitive to the implementation style of HTs. Experimental results show that* VeriTrust *is able to detect all HTs evaluated in this paper (constructed based on various HT design methodologies shown in the literature) at the cost of moderate extra verification time, which is not possible with existing solutions.*

1. INTRODUCTION

With globalization of the semiconductor industry, today's integrated circuit (IC) designs involve many third-parties during the design and manufacturing process, and hence they are vulnerable to a wide range of malicious alterations, namely hardware Trojans (HTs) [1]. For example, [2] reported a hardware backdoor found in a military grade FPGA, and King *et al.* [3] showed how easy it is to implement a HT in general-purpose processor, which grants privileged access to all memory elements of the system. Therefore, HTs are serious threats to military, financial, and other critical systems [4, 5].

HTs can be inserted in ICs in almost any stage, e.g., RTL design, logic synthesis, physical design, and manufacturing process. As it is not economically feasible to make the IC design and fabrication process completely trustworthy (even for military products), it is essential to develop verification techniques to tackle the challenging HT detection problem. Ideally, we would like to be able to detect a HT by activating it and observing its malicious behavior. In practice, however, since we are not knowledgeable about the location, the trigger condition and the malicious functionalities of the HT, it is very difficult, if not impossible, to directly activate it, especially considering that attackers would typically design a rare event to trigger the HT.

Most prior works on HT detection are based on *side-channel analysis* (*SCA*) (e.g., [6–10]). The idea behind is that a HT will affect some side-channel signatures, such as path delay and supply current, even if it is not functionally activated. A common assumption of these works is that HTs are inserted into some random ICs post-fabrication (instead of all) and there exists a trustworthy golden IC that has been thoroughly tested used for signature comparison or characterization. Consequently, these methods are not applicable to detect HTs inserted at design time, which will appear in every fabricated IC product.

Generally speaking, however, the likelihood of HTs being inserted at design time is much higher than that being inserted at manufacturing stage, because adversaries do not need to access foundry facilities to implement HTs. To the best of our knowledge, Hicks *et al.* [11] made the only attempt to detect HTs inserted at design time in the literature. Based on the observation that tricky HTs are usually not activated by test cases at design time (otherwise their malicious behavior will be already manifested), the HT detection problem can be formulated as how to identify "unused circuits" in the system.

However, there is no rigorous definition for "unused circuits" and hence the general *unused circuit identification* (*UCI*) problem is an open problem without clear solutions. In [11], the authors defined one type of unused circuits as follows. If any pair of related signals are equal throughout all test cases, the intervening circuits between them are regarded as unused ones and hence potential HTs, which can be replaced by a wire. Clearly, such restricted definition of unused circuits can only cover a small set of possible HTs. Later, [12, 13] presented how to automatically construct HTs that can evade the HT detection algorithm shown in [11].

Generally speaking, a HT is composed of its activation mechanism (referred to as *trigger*) and its malicious function (referred to as *payload*). In order to pass functional test and trust validations, stealthy HTs usually employ certain trigger condition that is controlled by dedicated trigger inputs and difficult to be activated with verification test cases. Based on this observation, in this paper, we propose a novel verification technique for hardware trust, namely *VeriTrust*, to identify the malicious trigger inputs for HT detection, by exploiting the fact that trigger conditions for HTs are not satisfied with verification test cases. The main contributions of this paper include:

- We classify HTs into two categories, *bug-based HTs* and *parasite-based HTs*, based on their impacts on the normal functionalities of the circuit, and discuss their corresponding characteristics.
- We present the so-called *VeriTrust* technique to detect parasite-based HTs by identifying the dedicated trigger inputs used in HTs. Unlike existing HT detection algorithms, *VeriTrust* is insensitive to the implementation style of HTs and hence prevents attackers from defeating it by simple HT modifications.
- We propose several techniques to reduce the memory usage and runtime of *VeriTrust* to make it scalable to large circuits.

The remainder of this paper is organized as follows. Section 2 presents preliminaries of this work. In Section 3, we characterize HTs inserted at design time into two types, namely *bug-based HTs* and *parasite-based HTs*. Next, we detail *VeriTrust* technique for detecting parasite-based HTs in Section 4. Experimental results are then presented in Section 5. Finally, Section 6 concludes this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'13, May 29-June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2071-9/13/05 $15.00.

2. PRELIMINARIES

2.1 Hardware Trust Challenges

Traditionally, the hardware layer of computing systems is often implicitly regarded as trustworthy. This assumption turns out to be quite naive [2–5], and several governments have expressed serious concerns about IC security [14, 15]. Designing HTs that are able to evade traditional IC verification tests is in fact not a very challenging task. This is because, the objective of these techniques (e.g., simulation and emulation) is to ensure an IC performs its specified functionalities. They do not intend to detect extra functionalities introduced into the design. Given the huge state-space that HTs can hide within a reasonably sized circuit, attackers can easily employ a trigger condition that has extremely low probability to be activated with verification tests. Various HT designs have been shown in the literature (e.g., [16, 17]) in recent years, and the Trust-Hub website [18] has released a set of HT benchmark circuits with different triggers and payloads.

Ideally, we would like to prevent HTs from ever being inserted into ICs or ever being triggered at run time. There have been some recent research efforts to achieve the above objectives via design obfuscation and/or isolation [19–22]. These solutions facilitate to mitigate some HT threats, but the associated design cost is quite high and there is no guarantee that ICs would be HT-free with these design methodologies.

2.2 Verification for Hardware Trust

Most existing HT detection techniques consider HTs inserted during fabrication and use side-channel analysis for HT detection (e.g., power-based analysis [6], timing-based analysis [7], and current-based analysis [8]). To reduce the sensitivity of SCA-based HT detection methods on the process variation, several *gate-level characterization* (GLC) techniques were proposed for HT detection recently [9, 10].

For HTs inserted at the design stage, they usually keep dormant when applying verification tests because otherwise their malicious behaviors would have manifested themselves. From this perspective, if part of circuits in a design is not sensitized with verification test cases, it is likely that such unused circuitry contains a HT inserted by attackers. Consequently, the HT detection problem can be formulated as an unused circuit identification problem [11]. *It is important to note that we can define many kinds of "unused circuits" and develop the corresponding UCI algorithms, but whether a particular UCI algorithm is effective or not depends heavily on the definition itself.*

One way to define "unused circuit" is based on the code coverage metrics used in verification. During circuit verification, we have a number of widely-used code coverage metrics: line coverage, condition coverage, toggle coverage, finite state machine (FSM) coverage, branch coverage, and path coverage. We can simply define "unused circuits" as uncovered parts with respect to code coverage metrics and focus on them for HT detection. The above metrics facilitate to identify some HTs, but attackers can easily defeat such HT detection techniques by coding RTL in a different style as detailed in Appendix B.

Hicks *et al.* [11] defined another type of "unused circuit" as follows. Consider a signal pair (s,t), where t is dependent on s. If $t = s$ with all verification test cases, the intermediate circuit between s and t is regarded as "unused circuit". With the above definition, the UCI algorithm traces all signal pairs during the verification and reports those for which the property $s = t$ holds throughout all test cases as places where potential HTs may lie.

For a HT whose payload is implemented separately from the circuit's normal function, the UCI algorithm presented in [11] is guaranteed to find it. This is because, there would exist a dedicated signal representing the circuit's normal function and it is equal to the final output of the circuit throughout all the verification test cases when the HT is not triggered. Because of this, [11] is able to detect some HTs that evade the earlier-mentioned coverage-oriented HT detection method. However, the fact that the effectiveness of [11] relies on HT implementation style enables the simple attack. That is, it is fairly easy to modify the implementation of the HT so that no signal pairs are equal to each other during verification (an example is given in Section 3.2), which has been shown in [12, 13].

Theoretically speaking, if we have a trustworthy high-level model of the design, we can resort to formal verification (FV) techniques [23] to check its equivalence with the questionable design for HT detection. In practice, however, FV techniques are usually not scalable to large circuits and the trusted high-level model may not be available.

2.3 Threat Model

As in [11], our threat model is that the hardware design can be covertly compromised by HTs inserted into the RTL code or the netlist, implemented by rogue designers in the design team or existing in third-party intellectual property cores to be integrated into the system. The motivation of HT insertion could be financial or general malice.

We assume the design verification procedure is not compromised and all verification test cases are trustworthy. We further assume a HT would be caught by verification test as long as it is triggered.

3. HT CLASSIFICATION

In this section, we classify HTs into two categories according to their impacts on the normal functionalities of the circuit. Attackers could either create certain HTs that directly modify the normal functionalities of the circuit, or insert certain HTs that introduce additional malicious behavior but keep all the normal functionalities of the circuit. They are named *bug-based HT* and *parasite-based HT* respectively in this paper.

For the ease of discussion, we have the following definitions:

DEFINITION 1. *A **functional input** is an input that is used by the circuit's specified normal functionality.*

DEFINITION 2. *A **trigger input** is an input that is used in the condition under which the HT is activated. Note that, functional inputs can serve as trigger inputs for HTs [12].*

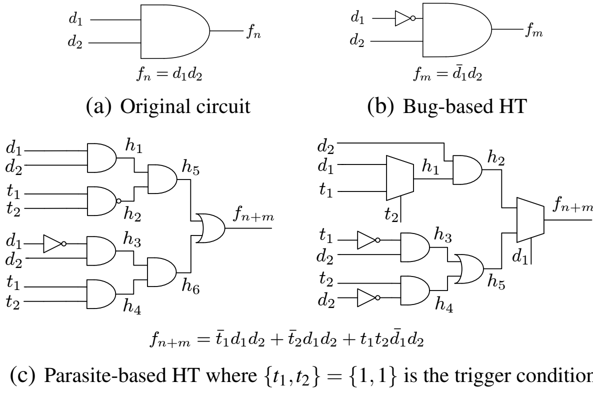
3.1 Bug-Based HT

A bug-based HT changes the circuit in a manner that causes it to lose some of its normal functionalities. Consider an original design in Fig. 1(a) whose normal function is $f_n = d_1 d_2$. An attacker may change it to a malicious function, $f_m = \bar{d}_1 d_2$, by adding an additional inverter, as shown in Fig. 1(b). With this malicious change, the circuit has lost certain functionalities, i.e., the two circuits behave differently when $d_2 = 1$. Their corresponding K-Maps are shown in Fig. 2(a) and Fig. 2(b). By comparing the two K-Maps, we can observe that some entries of the normal function have been modified by the malicious function as highlighted in gray. For bug-based HT, some functional inputs serve as trigger inputs to the HT, e.g., for the circuit shown in Fig. 1(b), d_2 is both a functional input and a trigger input.

From a different perspective, the bug-based HT can be simply regarded as a design bug (with malicious intention though), as the design in fact does not realize all of its normal functionalities designated by the specification. As a result, the extensive simulation/emulation is likely to detect this type of HTs. From this perspective, bug-based HT is usually not a good choice for attackers in terms of the stealthy requirement, and almost all HT designs appeared in the literature (e.g., [3, 11–13, 16–18]) belong to the parasite-based type, as discussed in the following.

3.2 Parasite-Based HT

A parasite-based HT exists along with the original circuit, and does not cause the original design to lose *any* normal functionalities. Again, consider an original circuit whose normal function is $f_n = d_1 d_2$. Suppose an attacker wants to insert a HT whose malicious function is $f_m = \bar{d}_1 d_2$ into the design. To control when the design runs the normal function and when it runs malicious function, the attacker could employ some additional inputs as trigger inputs, t_1 and t_2. In order to escape from trust validation, trigger inputs are usually carefully selected and the trigger condition is designed to be an extremely rare event occurred with verification tests.



(c) Parasite-based HT where $\{t_1, t_2\} = \{1, 1\}$ is the trigger condition

Figure 1: HT classification with a simple example.

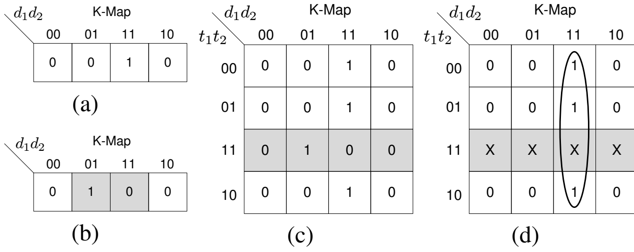


Figure 2: (a) K-Map of original circuit in Fig. 1(a); (b) K-Map of bug-based HT in Fig. 1(b); (c) K-Map of parasite-based HT in Fig. 1(c); (d) K-Map of parasite-based HT in Fig. 1(c) by setting entities of the malicious function as don't cares.

Let us examine the K-Map of the parasite-based HT-inserted circuit, as shown in Fig. 2(c). The third row represents the malicious function while other rows show the normal function. By comparing it with the K-Map of the original circuit (see Fig. 2(a)), we can observe that the parasite-based HT enlarges the K-Map size with additional inputs so that it can keep the original function while embedding the malicious function. The circuit can then perform the normal function and the malicious function alternately, controlled by trigger inputs.

With the above, we obtain the following two lemmas which we rely on to detect parasite-based HTs.

LEMMA 1. *Consider a signal that attackers expect to modify its value with a parasite-based HT, at least one dedicated trigger input must be employed to activate this parasite-based HT.*

LEMMA 2. *Suppose the value of a signal, denoted by* **S**, *can be manipulated by a parasite-based HT. Any signals that are logically-driven by signal* **S** *have at least one dedicated trigger input.*

The proofs of *Lemma 1* and *Lemma 2* are shown in Appendix A.

Lemma 1 enables us to focus on the HT trigger signal identification for HT detection. Then, with *Lemma 2* , it is not necessary to consider every signal in the circuit as HT-affected signal. Instead, we only need to consider inputs of state elements (e.g., flip-flops) and primary outputs of the circuit, which dramatically reduces the search space of our solution (detailed in Section 4).

Note that, parasite-based HTs may or may not be detected by the UCI technique presented in [11], depending on how the HT is implemented. Fig. 1(c) presents two possible implementations for the earlier example. The one at the lefthand side can be detected by [11], since f_{n+m} is equal to h_5 (the normal function) under all non-trigger conditions. The implementation shown at the righthand side, however, will evade [11], because the HT is not implemented separately from the circuit normal function and none of the signal pairs in the circuit are equal under all non-trigger conditions.

4. THE VERITRUST SOLUTION

In this section, we detail the proposed *VeriTrust* technique for detecting parasite-based HTs inserted at design time. For the convenience of presentation, HTs mentioned in the rest of the paper means parasite-based HTs unless otherwise specified.

4.1 Overview

Consider a signal whose driving combinational logic cone contains a HT. According to *Lemma 1*, attackers must employ at least one dedicated trigger input to manipulate the value of this signal. *VeriTrust* detects the HT by identifying such dedicated trigger inputs according to the following lemma.

LEMMA 3. *For a signal affected by a parasite-based HT, if we set all entries of the malicious functionalities of the HT as don't-cares, the dedicated trigger inputs used to activate HTs become redundant*[1].

The proof of *Lemma 3* is presented in Appendix A. Let us take the K-Map shown in Fig. 2(d) to illustrate *Lemma 3*. This K-Map represents the circuit in Fig. 1(c) where all entries of the malicious function are considered as don't-cares. By logic simplification, we obtain the original normal function to be $d_1 d_2$ and hence the dedicated trigger inputs, t_1 and t_2, become redundant.

Lemma 3 enables us to identify HT trigger inputs for a particular signal by setting entries of the malicious function as don't-cares. Such HT detection method has the advantage of being insensitive to the HT implementation style (when compared to existing HT detection techniques), because the entries that represent the malicious functionalities of a HT do not change with the implementations.

Although we cannot know which entry belongs to the malicious function a priori, what we do know is that these malicious entries must have not been activated during verification tests (otherwise the HT would have been detected already). In other words, any activated entry is HT-free, which is the premise of our *VeriTrust* solution.

Suppose the verification for the normal function of a circuit is complete, then the un-activated entries are composed of those unreachable entries in functional mode and entries from malicious function. Unreachable entries have no effects on the circuit outputs and hence can be safely ignored by setting them as don't cares. As a result, by setting all the un-activated entries as don't-cares, we can determine those redundant inputs as trigger inputs for HTs. However, in practice, verification tests are usually incomplete, which means un-activated entries may also belong to normal function. Under such circumstance, if we set all un-activated entries as don't-cares to identify redundant inputs, the found ones may include both functional inputs and trigger inputs, and designers need to further examine them to identify true HT trigger inputs, if any. It is important to note that, we may include some functional inputs as potential trigger inputs due to incomplete verification tests, but we will *never* miss any dedicated trigger inputs when they do exist. Therefore, *VeriTrust* guarantees to detect parasite-based HTs, which is not possible with existing HT detection techniques.

The earlier discussion focuses on combinationally-triggered HTs, now let us examine whether *VeriTrust* is able to detect those sequentially-triggered HTs. Generally speaking, there are two kinds of sequential triggers widely used in the literature: the counter-based trigger and the pattern-based trigger. They are typically implemented in a similar manner that one or more dedicated signals are asserted to trigger the HT when the counter reaches the pre-defined counter value or the specific trigger pattern appears [18]. Because these dedicated signals driven by the actual trigger inputs are redundant for the normal function within the combinational logic cone driving the HT-affected signal (according to *Lemma 3*), *VeriTrust* is capable to detect such HTs.

To sum up, consider a RTL design or a synthesized gate netlist that may contain HTs, our *VeriTrust* technique detects HTs by looking for redundant inputs after setting all un-activated entries during verification tests to be don't-cares. From this perspective, *Veritrust* can be

[1] An input is redundant if its value change has no impact on any circuit output.

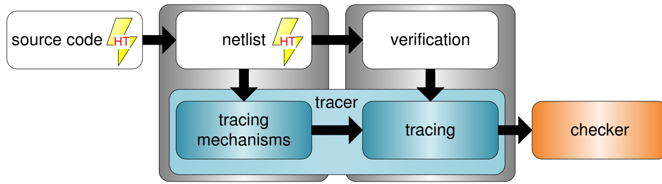


Figure 3: The overview of *VeriTrust*

considered as an "unused input identification" technique. The overall framework of *VeriTrust* is shown in Fig. 3, which contains two parts: the *tracer* and the *checker*. The tracer traces verification tests to identify those signals that contain un-activated entries by tracing mechanisms that are constructed according to the design netlist. According to *Lemma 2*, it is not necessary to consider every signal in the circuit for HT detection and the signals of interest during tracing are the inputs to all flip-flops and primary outputs. Then, the checker analyzes these signals and determines whether any of them indeed contain redundant inputs and hence are potentially affected by HTs. The details of the tracer and the checker are shown in the following subsections.

4.2 Tracer

Consider a particular signal whose fan-in logic cone may contain a HT, the responsibility of the tracer is to find out whether it contains any un-activated entries after verification tests. A straightforward method is to record the activation history of each and every entry, but it would require unaffordable memory space for large circuits and incur high runtime overhead. To resolve this problem, instead of tracing the activation history of each and every logic entry, we propose to trace in a much more compressed form.

Before discussing the details, let us revisit some basics of Boolean functions. In general, any combinational circuit can be represented in the form of sum-of-products (SOP) and product-of-sums (POS). SOP uses OR operation to combine those on-set minterms, while POS uses AND operation to combine those off-set maxterms. Two minterms (maxterms) are *adjacent* if they have only one different literal.

Next, let us define three new terms, *malicious on-set minterm*, *malicious off-set maxterm* and *dummy term* that compose malicious function as follows.

DEFINITION 3. *The **malicious on-set minterm** is the on-set minterm in the malicious function whose adjacent minterms in the normal function are off-set.*

DEFINITION 4. *The **malicious off-set maxterm** is the off-set maxterm in the malicious function whose adjacent maxterms in the normal function are on-set.*

DEFINITION 5. *The **dummy term** is the on-set minterm or off-set maxterm in the malicious function whose adjacent minterms or maxterms in the normal function are also on-set or off-set.*

With the above definitions, only the malicious on-set minterms and malicious off-set maxterms have malicious behavior. *Consequently, it is not necessary to set dummy terms as don't cares to indentify dedicated trigger inputs by* Lemma 3. Fig. 4 shows the K-Maps of two example HT-infected circuits (t_1 and t_2 are trigger inputs) to illustrate the above terms. The entry filled with vertical lines is malicious on-set minterm, as its neighboring entries in the normal function are all logic '0's. The entry filled with horizontal lines is malicious off-set maxterm, as its neighboring entries in the normal function are all logic '1's. The remaining entries without vertical lines and horizontal lines in the malicious function are dummy terms, as they have the same values with their neighboring entries in the normal function. From the above definitions, when simplifying the circuit into the minimal form of SOP (POS), we have the following observations:

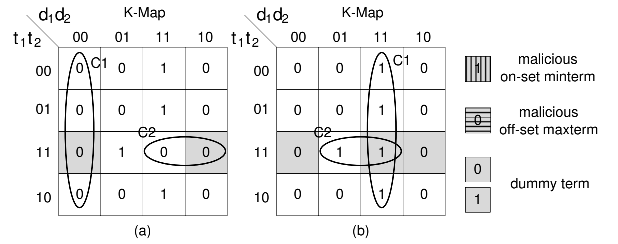


Figure 4: Two HT-affected circuits triggered by $\{t_1, t_2\} = \{1, 1\}$.

- Malicious on-set minterms and malicious off-set maxterms can only be combined with terms in the malicious function. This is because all the adjacent minterns (maxterms) of malicious on-set minterms (malicious off-set maxterms) in the normal function are off-set (on-set). For example, in Fig. 4 (a), one malicious off-set maxterm is combined with one dummy term (circled by C2), and in Fig. 4 (b), one malicious on-set maxterm is combined with one dummy term (circled by C2).
- Dummy terms can be combined with terms in the normal function as well as terms in the malicious function. For example, the circle C1 in Fig. 4 (a) and Fig. 4 (b) shows that the dummy term is combined with terms in the normal function; the circle C2 in Fig. 4 (a) and Fig. 4 (b) shows that the dummy term is combined with terms in the malicious function.

From the above, simplified products or sums containing malicious on-set minterms or malicious off-set maxterms cannot be activated during the verification. In other words, during the tracing process, we can record the activation history of products and sums instead of that of each logic entry, and hence the memory requirement and runtime overhead of our tracer can be dramatically reduced. The simplified products and sums can be obtained by simplifying the circuit to the minimum SOP and POS form, by leveraging the capability of logic synthesis tool. Note that, the proposed tracing methodology is applicable even if the circuit is not simplified to the minimum SOP and POS form. For the extreme case when we do not perform any simplification, the proposed tracer is simply degraded to tracing all entries.

The tracing procedure might still be time-consuming if the number of products and sums to be traced is large. To resolve this problem, we adopt the following two methods to reduce tracing overhead. First, we periodically remove those sums and products that have been activated. Second, we periodically remove those unactivated sums and products whose signal is determined to be HT-free with our checker.

4.3 Checker

The tracer outputs a number of signals that have un-activated products or sums after applying verification tests. The checker then checks whether a particular signal is driven by any redundant input by assigning the corresponding un-activated products and sums to be don't-cares. If it is, it would be a suspicious HT-affected signal; otherwise it is guaranteed to be HT-free.

To identify whether a signal is driven by redundant inputs could be time-consuming if its fan-in logic cone is large. To mitigate this problem, we adopt three methods for redundant input identification, which differ in the checking capability and time complexity.

Checker 1 simply checks whether there is any redundant input by removing un-activated products/sums from the signal's SOP/POS representation. Take the circuit in Fig. 4 (a) as an example. One SOP can be represented as: $f = d_1d_2 + t_1t_2d_2$. If we remove the un-activated $t_1t_2d_2$ from the SOP, the logic function becomes $f = d_1d_2$ with redundant inputs t_1 and t_2. This efficient checking mechanism is effective in many cases, but it cannot guarantee complete identification of redundant inputs. This is because, whether checker 1 can find out redundant inputs depends on the SOP/POS representation of the signal. For example, if the circuit in Fig. 4 (b) is represented as $f = \bar{t}_1d_1d_2 + \bar{t}_2d_1d_2 + t_1t_2d_2$, then removing the un-activated $t_1t_2d_2$ cannot leave t_1 and t_2 redundant.

Checker 2 leverages logic synthesis to re-simplify the function by considering un-activated products and sums as don't-cares. If an input does not appear in the synthesized circuit, it is a redundant input. This method is able to find most redundant inputs, but still cannot guarantee to find all since synthesis tool cannot guarantee optimality by employing heuristic algorithm for logic minimization.

Checker 3 verifies all inputs that are used in the un-activated products/sums one by one. If the change of an input would not cause the change of the function in all input patterns under the condition that un-activated products and sums are set as don't cares, it should be a redundant input. Checker 3 can guarantee to find out all redundant inputs, but it is more time-consuming than Checker 1 and Checker 2.

To ensure complete identification capability while keeping computational time low, we run checker 1, checker 2 and checker 3 in a consecutive manner. That is, if a more efficient checker (e.g., checker 1) finds out a redundant input for a particular signal with un-activated products/sums, we mark it as a suspicious HT-affected signal and move to process the next signal of interest. Otherwise, we use the next checker for redundant input identification. If all the three checkers cannot find redundant inputs for the signal of interest, it is guaranteed to be HT-free. Eventually, the checker returns a list of suspicious signals that are potentially affected by HTs.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

The HTs used in our experiments are obtained from two sources: the Trust-Hub website [18] and some related papers [3, 11–13, 16], detailed in Appendix B. We do not directly conduct experiments on those circuits in which HTs are originally inserted, because verification tests required by both [11] and *VeriTrust* for HT detection are not available. Instead, we have selected a SoC design from OpenCores [24], containing a 32-bit RISC microprocessor namely *OpenRisc* and many peripherals such as UART, USB and MAC, as the hardware platform for HT insertion and detection. We adopt the 17 test cases bundled with this design for verification.

For those RTL HTs, we carefully transplant them onto our experimental platform, keeping their triggers and payloads as discussed in Appendix B. For those HTs for circuit netlist, we use Synopsys Design Compiler to obtain the netlist of the design, and then we insert these HTs by copying gates used by the HT into the netlist and connect them to the targeted signals. To evaluate the effectiveness of *VeriTrust*, we compare it with [11] and code coverage metrics.

5.2 Results and Discussion

Generally speaking, a HT detection algorithm reports a list of candidate places where HTs may lie and requires designers to inspect further to identify whether these suspicious places indeed contain HTs. On one hand, if a true HT evades from the detection algorithm and does not appear in the candidate list, this would cause catastrophic effect; on the other hand, if the candidate list is very large, it will require much manpower to conduct further inspection. We therefore show results on the above two aspects first, and then present the runtime overhead of *VeriTrust*.

5.2.1 The Detection Capability

Fig. 5 presents the HTs identified by *VeriTrust* and UCI techniques after applying all the 17 test cases. Note that, a HT is considered to be detected if part of its trigger and/or payload is shown in the candidate list, reported by the corresponding HT detection algorithm.

First, let us look at the HT detection capability of UCI techniques for RTL HTs, as shown in Fig. 5(a). For HTs in G1, we observe that condition coverage detects T3 and T5 and FSM coverage detects T17 only, because most of the HTs in G1 are implemented without conditional expressions and FSMs. The line coverage, toggle coverage, branch coverage and path coverage detect all of them except T19, because T19 is implemented as a pure combinational logic block and all

Group	Index	Line	Cond	FSM	Toggle	Branch	Path	[11]	VeriTrust
G1	T1	√			√	√	√	√	√
	T3	√	√		√	√	√	√	√
	T5	√	√		√	√	√	√	√
	T7	√			√	√	√	√	√
	T9	√			√	√	√	√	√
	T11	√			√	√	√	√	√
	T13	√			√	√	√	√	√
	T15	√			√	√	√	√	√
	T17	√		√	√	√	√	√	√
	T19							√	√
	T21	√			√	√	√	√	√
	T23	√			√	√	√	√	√
G2	T2								√
	T4								√
	T8								√
	T10								√
	T12	√			√	√	√	√	√
	T14								√
G3	T6								√
	T16								√
	T18				√			√	√
	T20								√
	T22								√
	T24								√
Sum		12	2	1	13	12	12	14	24

√ means the HT is detected by this method.

(a)

	T25	T26	T27	T28	T29	T30	T31	T32	T33	Sum
[11]	√		√		√		√		√	5
VeriTrust	√	√	√	√	√	√	√	√	√	9

√ means the HT is detected by this method.

(b)

Figure 5: The summary of HTs identified by UCI techniques and *VeriTrust*

signals used by HTs have both 1-to-0 and 0-to-1 transitions. [11] detects all HTs in G1 because the final output is equal to the output of the normal function throughout the test cases. For HTs in G2, only T12 is detected by some code coverage metrics and [11]. This is because one signal partially indicating the trigger condition of T12 is quite difficult to be sensitized during verification and hence keeps constant. Theoretically speaking, however, HTs in G2 are likely to evade such UCI techniques, because all signals partially indicating the trigger condition could be set separately without activating the HT and there are no two signals that can hold equal under all non-trigger conditions. It is the fact that the lack of sufficient verification test cases exposes T12. The condition coverage and FSM coverage miss T12 because there are no conditions and FSMs in T12. Finally, for HTs in G3, T18 is detected by some code coverages and [11] due to the same reason as T12. Compared to T12, T18 evades line coverage, branch coverage and path coverage further. This is because T12 is implemented by the basic AND, OR and NOT operators.

From the above, we can observe that the different HT implementation style has a high impact on the detection capability of UCI techniques and none of them are able to detect all HTs. On the contrary, *VeriTrust* is insensitive to HT implementation styles and detects all the HTs based on the fact that they all contain dedicated trigger inputs.

Fig. 5(b) presents the HT detection result for HTs inserted into circuit netlist. As code coverage metrics are not meaningful for circuit netlist, we only present results for [11] and *VeriTrust*. It can be observed that *VeriTrust* can detect all the HTs while [11] misses T26, T28, T30 and T32 due to the fact that none of pairs of signals for HTs keep equal throughout the verification tests.

5.2.2 The Number of Suspicious Candidates

Fig. 6 presents the number of suspicious candidates reported by [11] and *VeriTrust*. By comparing the three curves, we observe that the number of suspicious candidates reported by *VeriTrust* is much smaller than those reported by [11] initially, especially when [11] is

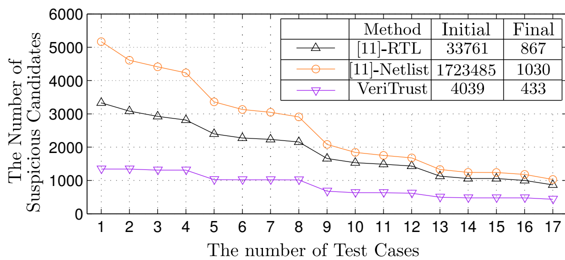


Figure 6: The number of suspicious candidates reported by [11] and *VeriTrust* with the increase number of test cases

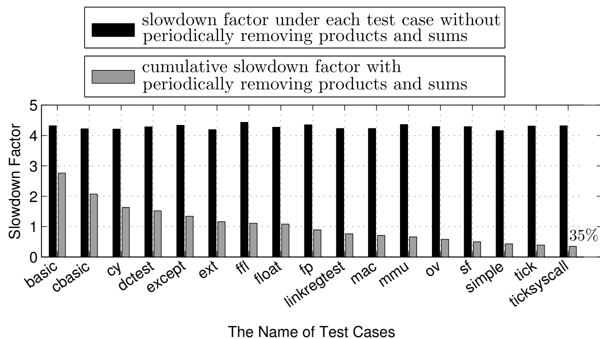


Figure 7: The slowdown factor of *VeriTrust* compared with the *base*

applied to the netlist. This is because the number of state elements traced by *VeriTrust* is much smaller than the number of signal pairs traced by [11] *VeriTrust*. On the other hand, the number of suspicious candidates reported by *VeriTrust* decreases slower. This is because, as soon as a signal pair is with different values, [11] can abandon it while *VeriTrust* requires to accumulate sufficient activated products and sums of the signal to deem it as HT-free.

Further trust validation is needed to determine whether the final list of suspicious candidates are indeed affected by HTs. As shown in Fig. 6, we can observe that the total number of suspicious candidates reported by *VeriTrust* is smaller than that reported by [11] but they are in the same order.

5.2.3 Runtime Overhead

Fig. 7 presents the slowdown factor of *VeriTrust* when compared against the case of functional verification without HT detection (referred to as *base*). It can be observed that the slowdown factor without periodically removing sums and products is quite high, about 4.25, as shown in the black bars. With periodically removing sums and products, however, the slowdown factors reduce with the application of more test cases (due to less traced signals). As shown in the gray bars, *VeriTrust* requires about 35% more runtime to finish 17 test cases.

Finally, Fig. 8 compares the runtime overhead of [11] and *VeriTrust* for netlist HTs. For the fair comparison, we remove signal pairs of [11] periodically as well. As can be seen, the runtime of [11] is larger than *VeriTrust* due to the high number of to-be-traced signal pairs.

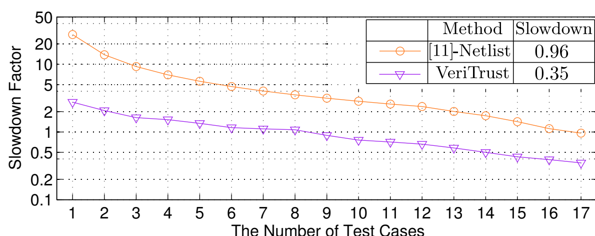


Figure 8: The slowdown factors of [11] and *VeriTrust* with the increase number of test cases compared with the *base*

6. CONCLUSION

In this paper, we propose a novel HT detection technique for HTs inserted at the design stage, namely *VeriTrust*, to automatically identify HTs trigger inputs by examining verification corners. The main advantage of *VeriTrust* compared to existing HT detection techniques is that *VeriTrust* is insensitive to HT implementation styles. Experimental results demonstrate that all the HTs that we have evaluated based on existing HT designs appeared in the literature are detectable with *VeriTrust*.

HT design and HT detection are like arms race, wherein attackers constantly update their tactics to intrude a system while defenders respond with more security measures to protect the system. We discuss the limitations of *VeriTrust* in Appendix C and we plan to investigate whether they can be used to evade *VeriTrust*.

7. ACKNOWLEDGEMENT

We thank Professor Srini Divadas for his insightful comments that greatly improved the paper. This work was supported in part by a CUHK Direct Grant No. 2050488.

8. REFERENCES

[1] M. Tehranipoor and F. Koushanfar. A survey of hardware Trojan taxonomy and detection. *IEEE Design & Test of Computers*, vol.27, no.1, 2010.

[2] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proc. International Conference on Cryptographic Hardware and Embedded Systems*, pp. 23–40, 2012.

[3] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proc. USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[4] J. Markoff. Old trick threatens the newest weapons. In *The New York Times*, p. D1, Oct. 27, 2009.

[5] S. Adee. The hunt for the kill switch. *IEEE Spectrum*, pp. 34–39, 2008.

[6] D. Agrawal, et al. Trojan detection using IC fingerprinting. In *Proc. IEEE Symposium on Security and Privacy*, pp. 296–310, 2007.

[7] J. Li and J. Lach. At-speed delay characterization for IC authentication and trojan horse detection. In *Proc. IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 8–14, 2008.

[8] D. Du, S. Narasimhan, R. S. Chakraborty, and S. Bhunia. Self-referencing: a scalable side-channel approach for hardware trojan detection. In *Proc. International Conference on Cryptographic Hardware and Embedded Systems*, pp. 173–187, 2010.

[9] Y. Alkabani and F. Koushanfar. Consistency-based characterization for IC trojan detection. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp. 123–127, 2009.

[10] S. Wei, S. Meguerdichian, and M. Potkonjak. Gate-level characterization: foundations and hardware security applications. In *Proc. ACM/IEEE Design Automation Conference*, pp. 222–227, 2010.

[11] M. Hicks, et al. Overcoming an untrusted computing base: detecting and removing malicious hardware automatically. In *Proc. IEEE Symposium on Security and Privacy*, pp. 159–172, 2010.

[12] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: building stealthy and malicious hardware. In *Proc. IEEE Symposium on Security and Privacy*, pp. 64–77, 2011.

[13] J. Zhang and Q. Xu. On Hardware Trojan Design and Implementation at RTL. *Proc. IEEE International Symposium on Hardware-Oriented Security and Trust*, to appear, 2013.

[14] U. S. Dept. of Defense. Defense Science Board Task Force on High Performance Microchip Supply. http://www.acq.osd.mil/dsb/reports/ADA435563.pdf, 2005.

[15] M. Beaumont, B. Hopkins, and T. Newby. Hardware trojans-prevention, detection, countermeasures (a literature review), 2011.

[16] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak. Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry. In *Proc. ACM/IEEE Design Automation Conference*, pp. 90–95, 2012.

[17] Y. Jin, N. Kupp, and Y. Makris. Experiences in hardware trojan design and implementation. In *Proc. IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 50–57, 2009.

[18] Trust-Hub Website. https://www.trust-hub.org/.

[19] R. Chakraborty and S. Bhunia. Security against hardware trojan through a novel application of design obfuscation. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pp. 113–116, 2009.

[20] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *Proc. IEEE Symposium on Security and Privacy*, pp. 49–63, 2011.

[21] A. Waksman and S. Sethumadhavan. Tamper evident microprocessors. In *Proc. IEEE Symposium on Security and Privacy*, pp. 173–188, 2010.

[22] T. Huffmire, et al. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Proc. IEEE Symposium on Security and Privacy*, pp. 281–295, 2007.

[23] S. Vasudevan, J.A. Abraham, V. Viswanath, and J. Tu, Automatic decomposition for sequential equivalence checking of system level and RTL descriptions. In *Proc. ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 71–80, 2006.

[24] OpenCores Website. http://opencores.org/.

APPENDIX

A. LEMMAS AND PROOFS

LEMMA 1. *Consider a signal that attackers expect to modify its value with a parasite-based HT, at least one dedicated trigger input must be employed to activate this parasite-based HT.*

PROOF. Suppose there exists a signal that is affected by a HT but does not have any dedicated trigger input. In this case, all inputs are functional inputs to this signal. Then, this HT must change the signal's normal function under some functional input patterns in order to be effective. Since any change would lead to the loss of original functionalities, it can only be a bug-based HT instead of a parasite-based HT. In other words, the parasite-based HT must be designed with at least one dedicated trigger input. ∎

LEMMA 2. *Suppose the value of a signal, denoted by **S**, can be manipulated by a parasite-based HT. Any signals that are logically-driven by signal **S** have at least one dedicated trigger input.*

PROOF. For any signal that is logically-driven by signal ***S***, its value can be manipulated by the parasite-based HT as well. Then, according to *Lemma 1*, it should have at least one dedicated trigger input in order to activate the HT. ∎

LEMMA 3. *For a signal affected by a parasite-based HT, if we set all entries of the malicious functionalities of the HT as don't-cares, the dedicated trigger inputs used to activate HTs become redundant.*

PROOF. Consider a signal affected by a parasite-based HT whose function could be represented by $f = C_n f_n + C_m f_m$, wherein f_n and f_m denote the normal function of the circuit and the malicious function of the HT, which are controlled by the non-trigger condition C_n and the trigger condition C_m, respectively. If we set all entries of the malicious function $C_m f_m$ as don't-cares, from the perspective of the design, we can use $C_m f_n$ to replace $C_m f_m$. Then, the new function becomes $f' = (C_n + C_m) f_n$, which means it exhibits the normal function f_n under both trigger and non-trigger conditions. As a result, dedicated trigger inputs become redundant. ∎

B. HARDWARE TROJANS USED IN EXPERIMENTS

The HTs used in our experiments are obtained from two sources: the Trust-Hub website [18] and some related papers [3, 11–13, 16], as summarized in Table 1. To be specific, there are 33 HTs, wherein 24 HTs are inserted into the HDL source code at the register-transfer level (RTL) while the other 9 HTs are inserted into netlist. We do not include other HTs appeared in the literature, because we can find their corresponding types from what we used after examination. We believe these HTs are sufficient to evaluate UCI techniques and the proposed *VeriTrust* technique thoroughly.

We rename these HTs as shown in the first column of Table 1. Column 2 presents the name of designs where HTs are originally inserted. Column 4 and Column 5 demonstrate the trigger and the payload of HTs, respectively. Based on their triggers and payloads, we have the following observations:

- The trigger of these HTs can be roughly classified into two categories: ❶ counter-based trigger and ❷ pattern-based trigger. For counter-based trigger, the counter can be pulsed by any signals (e.g., clock, instruction, data signal). For pattern-based trigger, the trigger observes a specific pattern or a sequence of specific patterns defined by the attacker. Signal driving the counter or the pattern are usually independent from the HT payload to lower the probability of the trigger being activated.
- The HT payloads have quite diverse malicious functionalities, e.g., compromising address or data register, leaking secrete information, reducing performance, depending on the objective of the attacker.

B.1 RTL HTs

T1–T20 are HT benchmark circuits in the form of RTL source code from Trust-Hub [18], originally inserted into three designs: MC8051, RISC, and RS232. As shown in [13], these HTs can be easily detected by both coverage metrics and [11], because all code lines controlled by the trigger condition of the HT that is indicated by a specific signal are never executed during the verification and the signal affected by the HT is always driven by one signal indicating the normal function.

While the above shows the effectiveness of UCI techniques for HT detection, it is still possible to make these HTs evade UCI techniques by simple HT design modifications. We therefore intentionally revise some HT implementations for comparison between UCI techniques and *VeriTrust* by using two code models proposed by [13]. Compared to the original HT implementation used by Trust-Hub, these two code models partition both the trigger condition and the normal function and adopt multiple signals indicating them. In this way, all code lines controlled by part of the trigger condition can be executed under non-trigger condition, and the signal affected by the HT can be driven by multiple signals alternately. The difference between two code models from [13] is that the trigger in the second code model is constructed with basic AND, OR and NOT operators.

Besides the 20 HTs from Trust-Hub, we use another 4 HTs from previous works. For T21 and T23 from [11], only high-level description on their triggers and payloads are available, and we choose to use the code model of HTs from Trust-Hub to construct them. T22 from [12] is a combinationally-triggered HT that can be mapped to the second code model in [13]. T24 is constructed by ourselves based on the malicious circuit provided by [12], and it is also a combinationally-triggered HT.

With the above, the 24 HTs in the form of RTL source code can be classified into three groups, G1, G2 and G3, based on the respective implementation methods.

- G1: HTs constructed with their original implementations whose indexes are odd, including T1,T3, T5, T7, T9, T11, T13, T15, T17, T19, T21 and T23.
- G2: HTs constructed with code model one in [13], including T2, T4, T8, T10, T12 and T14.
- G3: HTs constructed with code model two in [13], including T6, T16, T18, T20, T22 and T24.

B.2 HTs for Netlist

Among the 9 HTs for netlist, 8 are from Trust-Hub [18]. While there are a number of HT benchmarks for netlist shown in [18], their trigger mechanisms are quite similar and we select 8 HTs with different kinds of payloads. [11] can detect all the original HTs, because the final function is always the same as the normal function under all non-trigger conditions. Similarly, we select 4 HTs, T26, T28, T30 and T32 and do some modifications on them based on the code model of HTs for Netlist that is obtained by transferring the code model in [13] into the netlist. Therefore, theoretically, HTs constructed based on this code model can evade [11] in all non-trigger conditions.

In addition, we generate one rare switching HT based on [16], namely T33. In [16], Wei *et al.* developed the one-gate HT trigger to power on HT payloads. The main consideration of [16] when creating the HT is its leakage and timing impact on the design as it targets on the HT inserted at the post-fabrication. As we consider HTs inserted at design stage, we generate the one-gate HT trigger whose switching probability is the lowest among all gates, and we use this trigger to control the payload to change the value of one flip-flop.

C. LIMITATIONS OF VERITRUST

VeriTrust has been shown to be able to detect all the HTs that we have evaluated in this paper. However, this does not mean no HTs can defeat it. In particular, attackers may exploit the assumptions used in *VeriTrust* to evade it. We therefore discuss its limitations in this section.

Table 1: The summary of HTs used in our experiments

Index	Circuit	Level	Trigger	Payload
T1	MC8051-T200	RTL	❷ idle mode state	activate internal timer
T2	MC8051-T300	RTL	❷ specific data through UART	block receiving any message
T3	MC8051-T400	RTL	❷ a specific sequence of commands	disable interrupt
T4	MC8051-T500	RTL	❷ a specific sequence of commands	compromise received data
T5	MC8051-T600	RTL	❷ interrupt on INT0 pin	modify PC to disable jump
T6	MC8051-T700	RTL	❷ a specific command	compromise data
T7	MC8051-T800	RTL	❷ specific data through UART	manipulate stack pointer
T8	RISC-T100	RTL	❶ the number of specific instructions	change memory address
T9	RISC-T200	RTL	❶ the number of specific instructions	replace instructions with sleep command
T10	RISC-T300	RTL	❶ the number of specific instructions	transmit data to external storage
T11	RISC-T400	RTL	❶ the number of specific instructions	manipulate the address
T12	RS232-T100	RTL	❷ specific data through UART	stick a signal
T13	RS232-T200	RTL	❷ specific data	compromise performance counter
T14	RS232-T300	RTL	❶ transmitting time	compromise transmitted data
T15	RS232-T400	RTL	❷ specific data	compromise received data
T16	RS232-T500	RTL	❶ execution time	stick a signal
T17	RS232-T600	RTL	❷ a specific sequence of data	stick a signal & compromise data
T18	RS232-T700	RTL	❷ a specific sequence of data	stick a signal
T19	RS232-T800	RTL	❷ specific data from UART	manipulate the output signal
T20	RS232-T900	RTL	❷ specific data from UART	block the transmission
T21	Leon3 [11]	RTL	❷ a specific sequence of bus data	access protected memories
T22	Leon3 [12]	RTL	❷ a specific sequence of instructions	compromise the supervisor mode
T23	Leon3 [11]	RTL	❷ a specific sequence of bus data	execute arbitrary code
T24	OpenRisc	RTL	❶ a specific counter value	compromise a register
T25	s15850-T100	Netlist	❷ specific values of flip-flops	leak internal signal
T26	s35932-T100	Netlist	❷ specific values of flip-flops	enable the scan chain
T27	s35932-T200	Netlist	❷ specific values of flip-flops	mask four gates
T28	s35932-T300	Netlist	❷ specific values of flip-flops	slow down the path
T29	s38417-T100	Netlist	❷ specific values of signals	control an internal signal
T30	s38417-T200	Netlist	❷ specific values of signals	propagate erroneous value
T31	s38417-T300	Netlist	❷ specific values of signals	leak value through side-channel
T32	s38584-T200	Netlist	❶ a specific counter value	leak value to primary output
T33	OpenRisc	Netlist	❷ a specific data pattern	change the value of the flip-flop

❶: counter-based trigger ❷: pattern-based trigger

Firstly, *VeriTrust* is not able to detect bug-based HTs because it tries to find the trigger input that is redundant in terms of circuit normal functions. However, bug-based HTs are realized by using only functional inputs, thus bypassing *VeriTrust*. As discussed earlier, however, bug-based HTs can survive only in the hope of incomplete verification. Since attackers cannot control the design of verification test cases, the threat caused by bug-based HTs is usually small.

Secondly, *VeriTrust* would miss those HTs whose trigger is always on. This kind of HTs is usually used to compromise parameters of the design (e.g., timing, power or reliability) without introducing new functionalities to the design. For example, a HT may simply introduce some extra inverters on a circuit path to increase its delay. As *VeriTrust* focuses on detecting HTs that have function-level malicious behavior, these types of HTs can evade it. On the other hand, it is usually difficult to insert HTs to modify the circuit parameters at the design stage, because computer-aided design tools used in the later design stage (e.g., logic synthesis and physical design) may remove such impact with circuit optimization.

Finally, similar to [11], we assume that a HT is detected as long as it is functionally-activated. In practice, however, the verification test cases may miss identifying the malicious behavior of HTs. Therefore, it would be beneficial to review and/or redesign verification test cases for trust validation from this perspective. On the other hand, this problem is a less concern because attackers usually would not bet on careless verification to hide HT payloads.