On Hardware Trojan Design and Implementation at Register-Transfer Level

Jie Zhang 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, qxu}@cse.cuhk.edu.hk

***Abstract*—There have been a number of hardware Trojan (HT) designs at register-transfer level (RTL) in the literature, which mainly describe their malicious behaviors and trigger mechanisms. Generally speaking, the stealthiness of the HTs is shown with extremely low sensitization probability of the trigger events. In practice, however, based on the fact that HTs are not sensitized with verification test cases (otherwise their malicious behaviors would have manifested themselves), designers could focus on verification corners for HT detection. Consequently, a stealthy HT not only requires to be hard to trigger, but also needs to be able to evade those hardware trust verification techniques based on "unused circuit identification (UCI)". In this paper, we present new HT design and implementation techniques that are able to achieve the above objectives. In addition, attackers would like to be able to control their HTs easily, which is also considered in the proposed HT design methodology. Experimental results demonstrate that HTs constructed with the proposed technique are both hard to be detected and easy to be controlled when compared to existing HTs shown in the literature.**

I. Introduction

Today's integrated circuit (IC) designs usually involve large design teams and many third-parties. Consequently, they are vulnerable to a wide range of malicious alterations that subvert or compromise the normal operation of infected devices, namely hardware Trojans (HTs) [1]. For instance, a backdoor inserted into the Actel/Microsemi chips was recently found [2], and a report released by U.S. Senate Armed Services Committee claimed that about one million suspected bogus parts have been found in U.S. military aircraft [3]. Consequently, HTs have attracted serious attention from authorities [4], [5], which motivate a large amount of research efforts.

A. Related Work

HTs can be inserted into an IC at various design stages, e.g., specification, register-transfer level (RTL), layout and fabrication. Among them, HTs inserted at RTL by rogue designers in the design team or integrated into the system with third-party intellectual property (IP) cores are the most serious threats because attackers have high flexibility to implement any malicious function.

HT Detection: Detecting HTs inserted at RTL is extremely difficult since traditional IC verification techniques are not suitable for finding "extra" functionalities beyond circuit specification, especially considering the fact that attackers usually employ a rare event with extremely low activation probability in normal functional mode to trigger HTs. To the best of our knowledge, Hicks *et al.* [6] made the only attempt to detect HTs at RTL in the literature. Leveraging the fact that a HT usually keeps dormant during verification, the HT detection problem is formulated as an "unused circuit identification (UCI)" problem. That is, if part of circuits in a design is not sensitized with verification test cases, it is likely that such circuitry belongs to a HT. However, how to define "unused circuit" is not quite clear and the authors considered the following: if any pair of related signals are equal throughout all test cases, the intervening circuits between them are regarded as unused circuitry and hence potential HT. From another perspective, those uncovered parts with respect to code coverage metrics[1] during verification can be treated as another type of unused circuits. While UCI techniques are able to detect many existing HT designs shown in the literature to be detailed in Section II, they are sensitive to the actual HT implementation style [7].

HT Design and Implementation Methodology: HT detection and HT desigdesign are like arms race, wherein defenders constantly update their security measures to protect the system while attackers respond with more tricky HTs to intrude a system. There are a variety of HTs proposed in earlier works, and however they tend to ignore the importance of HT implementation. [8] developed two HTs compromising memory access mechanism and shadow mode mechanism to support software attacks without showing detailed implementation. [9], [10] presented several HT designs with various trigger methods and malicious behaviors in the embedded system challenge competition. [11] designed two simple HTs controlled by a counter for the RSA encryption circuit. [12] designed the HT to facilitate side-channel attack. However, none of them [8]–[12] provides detailed HT implementation. Recently, *Trust-Hub* website [13] has released a list of HT benchmarks, but almost all of them could be detected by UCI techniques as shown in Table I (see Section II). This is because these HTs are not carefully designed to be hidden as "useful circuits". Sturton *et al.* [7] designed some HTs that are able to evade [6]. However, their HT design and implementation methodology, on one hand, is based on exhaustive search to locate malicious circuitry that could be used to build HTs rather than a general HT design methodology, and on the other hand, it does not consider the traditional verification as well as code coverage metrics.

[1] Widely used code coverage metrics are line coverage, condition coverage, toggle coverage, FSM coverage, branch coverage and path coverage.

B. Summary of Contribution

Motivated by above, in this paper, we propose a systematic HT design and implementation methodology, aiming at making HTs bypass existing HT detection techniques without losing any flexibility of the HT. The approach designs and implements HTs from three aspects. First, to evade traditional verification tests, HTs keep dormant during verification through the selected rare trigger condition. Second, to evade UCI techniques, HTs are hidden as "useful circuit" with the proposed two HT coding models. Third, HTs should be as controllable as possible so that attackers could easily employ them to perform malicious operations.

The remainder of this paper is organized as follows. In Section II, we motivate this paper. Then, we discuss the proposed HT design and implementation methodology in Section III. Experimental results are presented in Section IV. Finally, we conclude this paper in Section V.

II. Motivation

Index	Circuit	TV	Line	Cond	FSM	Toggle	Branch	Path	[6]
T1	MC8051-T200		√			√	√	√	√
T2	MC8051-T300		√		√	√	√	√	√
T3	MC8051-T400		√	√		√	√	√	√
T4	MC8051-T500		√	√		√	√	√	√
T5	MC8051-T600		√	√		√	√	√	√
T6	MC8051-T700		√	√		√	√	√	√
T7	MC8051-T800		√			√	√	√	√
T8	RISC-T100		√			√	√	√	√
T9	RISC-T200		√			√	√	√	√
T10	RISC-T300		√			√	√	√	√
T11	RISC-T400		√			√	√	√	√
T12	RS232-T100					√	√		√
T13	RS232-T200		√			√	√	√	√
T14	RS232-T300		√			√	√	√	√
T15	RS232-T400		√			√	√	√	√
T16	RS232-T500		√			√	√	√	√
T17	RS232-T600		√		√	√	√	√	√
T18	RS232-T700		√		√	√	√	√	√
T19	RS232-T800								√
T20	RS232-T900		√		√	√	√	√	√

√ means the HT is detected by this method.

TABLE I: The summary of the HTs from Trust-Hub [13] identified by code coverage metrics and [6]

```
always @ (posedge clk)
    if (pattern ==PATTERN /
        counter == COUNTER)
        t_en <= 1'b1;
    else t_en <= 1'b0;
always @ (posedge clk)
    if (t_en == 1'b1) f <= f_m;
    else f <= f_n;
```

Fig. 1: The code model of most HTs from the Trust-Hub [13] written in Verilog HDL

Table I summarizes the HTs from Trust-Hub [13] identified by traditional verification tests denoted as *TV* and UCI techniques. The experiment is conducted on a SoC-based platform detailed in Section IV rather than on the original circuits, because there is no test case available for the original circuits. HTs are carefully transferred into the platform by connecting them on expected signals without any modification on the HT implementation. As can be seen, traditional verification tests miss all HTs while UCI techniques, especially [6], can detect all HTs. The reason is that these HTs are not hidden as "useful circuits". After further examination, we find that implementations of most HTs from the Trust-hub can be summarized as Fig. 1. One signal, denoted as t_{en}, is used to indicate the occurrence of the trigger condition, which is driven by either a specific pattern or a counter. This HT implementation can evade the traditional verification as long as the trigger condition is not activated, which is easily achieved with the huge state space of the circuit. However, it can be detected by UCI techniques, because code lines, "$t_{en} <= 1'b1$" and "$f <= f_m$", would not be executed and "f" would be always equal to "f_n" under all non-trigger conditions.

The above motivates us to propose a systematic HT design and implementation methodology to evade all existing HT detecting techniques.

III. HT Design and Implementation Methodology

Generally speaking, a HT is composed of its activation mechanism (referred as *trigger*) and its malicious function (referred as *payload*). Our objective is to design and implement HTs that can evade both the traditional verification and UCI techniques without sacrificing much flexibility. To achieve it, we consider three design and implementation rules. Firstly, HTs should keep dormant during the verification so as to evade the traditional verification. Secondly, HTs should be hidden as "useful circuit" so as to evade all UCI techniques. Thirdly, to make HTs flexible for attackers to perform malicious operations, it should be as controllable as possible. Since the payload mainly depends on the objective of attackers and also does not affect the stealthiness and the controllability of the HT, we only focus on the trigger design and implementation in this paper.

A. Rule One

Traditional verification and testing detects a HT by triggering it, and hence how to design a rare trigger condition that makes the HT keep silent during the verification becomes a key problem. To achieve it, attackers can generally employ the following four methods.

- Attackers can select trigger inputs whose trigger values are difficult to be sensitized during the verification.
- Attackers can adopt multiple trigger inputs. If there are l trigger inputs denoted as $t_1, t_2, \ldots, t_l$, the probability of the occurrence of the trigger condition is roughly equal to $\prod_{i=1}^{l} P_{t_i}$, wherein P_{t_i} is the probability of the trigger input i being the trigger value. However, considering the size of the HT, attackers can only use limited trigger inputs.
- Attackers can adopt a sequence of trigger values. Consider l trigger inputs, $t_1, t_2, \ldots, t_l$, and suppose attackers construct the trigger condition by m continuous trigger values. In this way, the probability of the trigger condition can be further reduced, represented as $\prod_{j=1}^{m} \prod_{i=1}^{l} P_{t_{ij}}$. Similarly, the number of continuous trigger values cannot be too large in the consideration of the hardware cost.

- Attackers can select trigger inputs from components that are less dependent, reducing the probability of the occurrence of trigger values at the same time. This is because the test case used by the designer usually targets on one or some specific normal functionalities.

Given the huge state-space that the HT can hide within a reasonably sized circuit, attackers can easily select some trigger inputs to construct the trigger condition. Consequently, rule one can be illustrated by minimizing the probability of the trigger condition subject to the hardware cost constraint.

$$\begin{aligned} \text{Objective :} \quad & \text{Min} \prod_{j=1}^{m} \prod_{i=1}^{l} P_{t_{ij}}; \\ \text{Constraint :} \quad & cost(m,l) < COST_{ex}. \end{aligned} \tag{1}$$

The hardware cost of HT is a function of the number of trigger inputs and the number of continues trigger values. Since the HT is implemented by the proposed code models discussed in rule two, we can estimate the HT cost by synthesizing code models in advance. Note that, the trigger condition in Eq. 1 ignores the dependency of signals, because obtaining accurate probability of the trigger condition through simulation is very time-consuming in the iterative algorithm.

Despite the rare trigger condition designed by rule one, HTs, however, could be detected by UCI techniques, as shown in Table I. We would discuss how to resolve this problem in rule two.

```
always @ (posedge clk) begin
    if (rst == 1'b1) counter_1 = 0;
    else counter_1 = counter_1 + 1;
    if (rst == 1'b1) counter_2 = 0;
    else counter_2 = counter_2 + 1;
    ⋮         ⋮         ⋮
    if (rst == 1'b1) counter_k = 0;
    else counter_k = counter_k + 1;
end
```

(a) Code model of the counter-based trigger

```
always @ (posedge clk) begin
    case(state)
        s_1 : if(pattern_1 == p_(1,1))
                  state <= s_2;
              else  state <= s_1;
        s_2 : if(pattern_1 == p_(2,1))
                  state <= s_3;
              else  state <= s_1;
        ⋮         ⋮         ⋮
        s_k : if(pattern_1 == p_(k,1))
                  state <= s_k+1;
              else  state <= s_1;
        default: state <= s_1;
    endcase
    if (state == s_k+1)
        t_en_1 <= 1'b1;
    else  t_en_1 <= 1'b0;
end
```

(b)Code model of the pattern-based trigger

```
always @ (posedge clk) begin
    if (pattern_1 == PATTERN_1 |
        counter_1 == COUNTER_1)
        t_en_1 <= 1'b1;
    else t_en_1 <= 1'b0;
    if (pattern_2 == PATTERN_2 |
        counter_2 == COUNTER_2)
        t_en_2 <= 1'b1;
    else t_en_2 <= 1'b0;
    ⋮         ⋮         ⋮
    if (pattern_k == PATTERN_k |
        counter_k == COUNTER_k)
        t_en_k <= 1'b1;
    else t_en_k <= 1'b0;
end
always @ (posedge clk) begin
    f <= (t_en_1 & t_en_2 & ··· & t_en_k) & f_m |
         (~ t_en_1 | ~ t_en_2 | ··· | ~ t_en_k)
         & (c_1 & f_n_1 | c_2 & f_n_2);
end
```

(c) Code model of the HT

Fig. 2: Code model One to implement the HT written in Verilog HDL

B. Rule Two

The main idea behind to make HT evade UCI techniques is to hide it as the "useful circuit" with respect to the definition of the "unused circuit". To achieve this, we propose a novel solution that combines the code writing style and the trigger input selection. As observed in the literature, there are mainly two kinds of triggers: *pattern-based trigger* and *counter-based trigger*, and thus we take them as examples to illustrate the proposed code models.

Let us start with the code writing style. We propose two code models for HT implementation based on the pattern-based trigger and counter-based trigger. Code model one is shown in Fig. 2. Compared to the code model used by HTs from the Trust-Hub [13] shown in Fig. 1, code model one has three differences. First, since code lines, conditions, FSMs, transitions of signals, branches and paths controlled by the trigger condition are definitely uncovered (e.g., Fig. 1), we propose to partition the trigger condition into multiple parts of the trigger condition, namely *sub-trigger condition*. In this way, all parts of the HT controlled by sub-trigger conditions can be covered, because sub-trigger conditions could be satisfied under certain non-trigger conditions. As shown in Fig. 2, we employ multiple patterns denoted as $pattern_1$, $pattern_2$, ..., and $pattern_k$ or multiple counters denoted as $counter_1$, $counter_2$, ..., and $counter_k$, and use multiple signals denoted as t_{en_1}, t_{en_2}, ..., and t_{en_k} to indicate the occurrence of corresponding sub-trigger conditions. Fig. 2 (a) and Fig. 2 (b) show how to implement a sub-trigger condition that is realized by a specific counter and a part of pattern. Second, since [6] is able to detect the HT whose affected output is always driven by the normal function, we partition the normal function into two sub-normal functions to make the final output be driven by sub-normal functions alternately. As shown, sub-normal functions denoted as f_{n_1} and f_{n_2} are controlled by their corresponding conditions denoted as c_1 and c_2. Third, the final output (f) composed of two sub-normal functions, malicious function, and their corresponding conditions are integrated by AND, OR, and NOT operators instead of "if-else" or "case" operators, and in this way, the assignment statement of the final output with the malicious function must be executed under non-trigger conditions.

Code model two, based on code model one, replaces all "if-else" and "case" operators in the trigger by AND, OR and NOT operators. Compared to code model one, HTs implemented by code model two can definitely evade condition coverage, FSM coverage, branch coverage and path coverage, because no conditions, FSMs, branches and paths are used. However, code model two would introduce more code lines for the HT, which is likely to attract the attention of designers [8].

The HT designed according to the two proposed code models can evade all existing UCI-based techniques as long as two conditions are satisfied during the verification: (1) t_{en_1}, t_{en_2},..., and t_{en_k} have both 0-to-1 and 1-to-0 transitions. (2) c_1 and c_2 have been set to logic '1' once. If all non-trigger conditions are sensitized, such two conditions can definitely be satisfied. However, it is nearly impossible to make all non-trigger conditions be verified with limited verification time and effort, which could cause HTs to be detected by UCI techniques. For instance, in Fig. 2, it is possible that part of the trigger condition (e.g., "$pattern1 == PATTERN1$") has never been satisfied, and hence the code line, "$t_{en_1} <= 1'b1$", is not covered and the signal, t_{en_1}, does not have both transitions,

which cause the whole HT to be detected by line coverage, toggle coverage and [6]. Similarly, the final output denoted as f could be driven by only one part of normal circuit (e.g., f_{n_1}), which makes HTs be identified by [6].

Consequently, HTs should be able to evade all UCI techniques as likely as possible even when the verification test cases are not complete. To achieve this, we are required to carefully select trigger input and partition trigger condition to increase the probability of each sub-trigger condition. Since whether HTs can evade UCI techniques is bounded by the least-likely-triggered sub-trigger condition and less-likely-occurring sub-normal function, we obtain the HT by maximizing probabilities of the least-likely-triggered sub-trigger condition and less-likely-occurring sub-normal function with the constraint of the number of code lines (NL_{ex}), given as:

$$\begin{aligned} \text{Objective}: &\ \text{Max}(\min\{P_{c_1}, P_{c_2}\}); \\ &\ \text{Max}(\min\{P_{t_{en_1}}, P_{t_{en_2}}, \ldots, P_{t_{en_k}}\}); \\ \text{Constraint}: &\ NL(k) < NL_{ex}; \end{aligned} \quad (2)$$

where P_{c_i} and $P_{t_{en_i}}$ denote probabilities of the occurrence of the i-th sub-normal function and the i-th sub-trigger condition, and $NL(k)$ denotes the number of code lines for HTs. $NL(k)$ is a linear function of the number of sub-trigger conditions (k) whose coefficients can be obtained from HT code models.

Maximizing the probability of the less-likely-occurring sub-normal function is relatively easy, because it is irrelevant to the trigger and payload design and designers would verify normal functionalities of the circuit as completely as possible. However, maximizing the probability of the least-likely-triggered sub-trigger condition can conflict with minimizing the probability of the trigger condition shown in Eq. 1. Therefore, how to select trigger inputs to not only keep the HT silent but also evade UCI techniques is challenging. We will discuss that in details in Section III.D.

C. Rule Three

Generally, attackers expect the HT designed to be as controllable as possible, increasing the flexibility to perform malicious operations. However, some signals cannot be controlled in terms of attackers (e.g., lacking in the physical access), which hence restricts choices of trigger inputs.

To design flexible HTs, we introduce the *un-controllability* of each signal that reflects the difficulty of setting a signal to a required value from the perspective of attackers. The un-controllability of each signal is defined as the number of signals that should be manipulated but cannot be manipulated to set a value of this signal from prime inputs, which includes the *0-un-controllability (UC0)* and *1-un-controllability (UC1)*.

We obtain the UC0 and UC1 of each signal by analyzing the circuit netlist. First, we set UC0 and UC1 of each prime inputs for the circuit according to the actual environment. If I, a prime input, cannot be controlled by the attacker, we have

$$UC1(I) = 1, \text{ and } UC0(I) = 1; \quad (3)$$

otherwise,

$$UC1(I) = 0, \text{ and } UC0(I) = 0. \quad (4)$$

With given UC0 and UC1 of each prime input, the UC0 and UC1 of each signal is calculated in the topology order from prime inputs to prime outputs. Fig. 3 presents the calculation of the un-controllability of each gate. Note that, the calculation

Symbol	Un-Controllability Calculation
a b c	$UC0(c) = min(UC0(a), UC0(b))$ $UC1(c) = UC1(a) + UC1(b)$
a b c	$UC0(c) = UC0(a) + UC0(b)$ $UC1(c) = min(UC1(a), UC1(b))$
a b c	$UC0(c) = min(UC0(a) + UC0(b), UC1(a) + UC1(b))$ $UC1(c) = min(UC0(a) + UC1(b), UC1(a) + UC0(b))$
a c	$UC0(c) = UC1(a)$ $UC1(c) = UC0(a)$

Fig. 3: The un-controllability calculation of each gate

of the un-controllability of each gate ignores the dependence of input signals for simplification.

With the un-controllability of each signal, attackers can determine trigger inputs with respect to the actual requirement. Suppose attackers choose $t_1, t_2, \ldots, t_l$ as trigger inputs and select their m specific continuous values together as the trigger condition. Then, the un-controllability of the trigger condition is the sum of un-controllability of each trigger input driven by corresponding trigger value, denoted as UC_T, which indicates that the number of signals that cannot be controlled to set the trigger condition by the attacker.

D. Overall Flow

As discussed, the above three rules have quite different requirements on the trigger input selection, and therefore how to consider them together in an HT design and implementation is a challenging problem.

In this paper, we emphasize rule two over rule one and rule three. Consequently, the "best" HT is obtained by maximizing the probability of the HT evading UCI techniques with the constraints of the probability of the trigger conditions ($P_{T_{ex}}$), the hardware cost ($COST_{ex}$), the number of code lines (NL_{ex}) and un-controllability of the HT ($UC_{T_{ex}}$), represented as:

$$\begin{aligned} \text{Objective}: &\ \text{Max}(\min\{P_{t_{en_1}}, P_{t_{en_2}}, \ldots, P_{t_{en_k}}\}); \\ \text{Constraints}: &\ \prod_{j=1}^{m}\prod_{i=1}^{l} P_{t_{ij}} < P_{T_{ex}}; \\ &\ cost(m,l) < COST_{ex}; \\ &\ NL(k) < NL_{ex}; \\ &\ UC_T < UC_{T_{ex}}. \end{aligned} \quad (5)$$

We do not maximize probabilities of less-likely-triggered sub-normal functions in Eq. 5, because it can be done independently without listed constraints.

Due to the huge solution space for the trigger condition, we solve the above problem by a heuristic algorithm described in Algorithm 1. In the beginning, as attackers, we obtain the probability of each signal by the simulation with guessed test cases as well as un-controllability of each signal (Line 1-2). To reduce the solution space of the trigger condition, we first determine the number of trigger inputs and the number of sequence of trigger values according to constraints of the hardware cost and the number of code lines (Line 3). Then, we sort all signals according to the larger probability between

Algorithm 1: Overall Flow

1 Run the simulation with guessed test cases to obtain the probability of each signal;
2 Calculate un-controllability of each signal;
3 Determine the number of trigger inputs (*l*) and the length of sequence of trigger values (*m*) according to constraints of hardware cost and the number of code lines;
4 Sort signals according to the larger probability between being logic '1' and logic '0' in the decreasing order;
5 **while** *the number of signals is larger than l and the number of recorded solutions is smaller than N* **do**
6 | Implement the HT with the first *l* signals in the signal list as trigger inputs and partition the trigger condition evenly;
7 | **if** *both the probability of the trigger condition and un-controllability of HT are satisfied* **then**
8 | | Record the current solution as a candidate;
9 | **end if**
10 | Remove the first signal in the signal list;
11 **end while**
12 Run simulation and determine the final solution from candidates;

being logic '1' or logic '0' in the descending order (Line 4). After that, the algorithm goes into a loop (Line 5-11). In each loop, we implement the HT by treating the first *l* signals in the signal list as trigger inputs and partitioning the trigger condition evenly. If both the probability of the trigger condition and un-controllability of HT are satisfied, we record the current solution. Since we choose *l* signals that have largest signal probabilities in the signal list each time, it is more likely to build sub-trigger conditions with high probabilities. Finally, we remove the first signal in the signal list and go back. The loop stops if the number of signals is smaller than *l* or the number of recorded solutions is larger than user-defined parameter *N*. The reason why we find a number of solutions is that the probabilities of the trigger condition and sub-trigger conditions are not very accurate based on the simple analysis. Consequently, in order to obtain a good HT, we run the simulation again to verify the designed HT, and select the solution that is the most likely to evade the UCI techniques among candidates.

IV. Experimental Results

A. Experimental Setup

We use 24 HTs in our experiments, wherein 20 are HT benchmark circuits from Trust-Hub [13] and the remaining 4 are from previous work [7], [8]. We believe these HTs are enough to evaluate the proposed HT design and implementation methodology. We keep their payloads but re-design their triggers by the proposed method.

We do not directly conduct experiments on those circuits in which HTs are originally inserted, because verification tests required by both code coverage metrics and [6] for HT detection are not available. Instead, we have selected a SoC design from OpenCores [15], 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. We assume attackers know 5 test cases among all for HT design while designers adopt the remaining 12 test cases to verify the design. During the HT design process, we set 1% hardware cost and 5% code line constraints and consider that prime inputs connecting the main memory, UART, USB and MAC can be controlled by attackers. Finally, HTs are carefully designed and inserted into the platform and controlled by different trigger conditions.

B. Results and Discussion

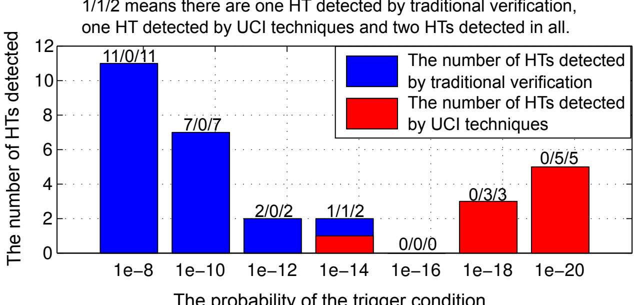
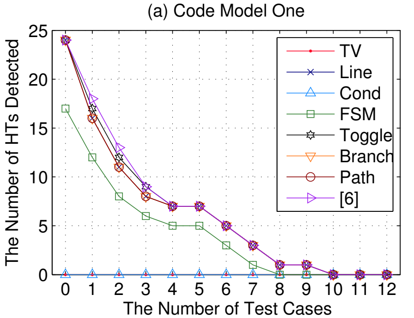


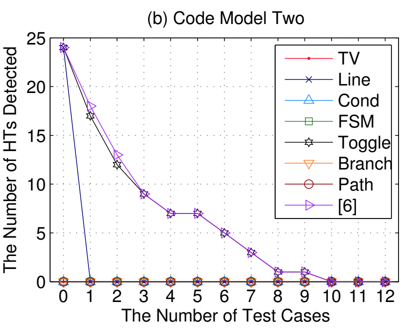
Fig. 5: The number of HTs detected by the traditional verification and UCI techniques under the constraint of diverse probabilities of the trigger condition

First, we study the effectiveness of the proposed HT design and implementation method. We adopt the traditional verification denoted as *TV* and 7 UCI techniques that are line coverage, condition coverage, FSM coverage, toggle coverage, branch coverage, path coverage and [6] to evaluate designed HTs, and results are shown in Fig. 4. For the traditional verification, we find that it misses all HTs since none of them have been activated during the verification. For UCI techniques, we consider a UCI technique detects an HT if it reports any part of the circuit that belongs to the HT. Therefore, without verification, each UCI technique detects all HTs that they cover. As seen, condition coverage covers none of HTs, because we do not use any conditions in the two code modes. Moreover, for code model one, FSM coverage does not cover all of HTs because some of them are counter-based HTs written without FSMs. Compared to code model one, none of HTs written by code model two are covered by FSM coverage, branch coverage and path coverage. This is because all FSMs, branches and paths are replaced with AND, OR and NOT operators that cannot be recognized by these coverage metrics. With test cases used in the verification, most methods begin to miss some HTs, because all parts of these HTs that UCI techniques focus on are verified. In the end, HTs designed by both code model one and two evade all UCI techniques. As shown in Fig. 4 (a) and Fig. 4 (b), it is possible to use fewer test cases to detect more HTs, but the fewer test cases would results in more wrongly-identified HTs to be reported, which leads to much more manual effort to further examination. We present the coverage[2] of each UCI technique in Fig. 4 (c). By comparing to original HTs without any modifications shown in Table I, HTs revised by proposed method are much more likely to evade existing HT detection methods.

Next, we present the impact of the probability of the trigger condition on designed HTs. The result is shown in Fig. 5. As can be observed, with the decrease of the probability of the

[2]The coverage of an UCI technique is the proportion of candidates that are verified by this technique.





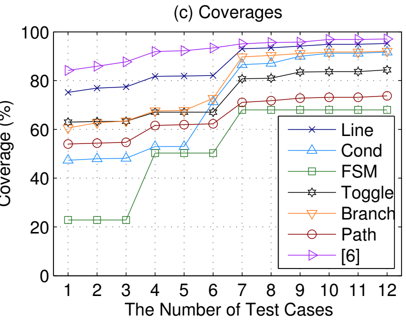


Fig. 4: (a) and (b) present numbers of HTs detected by code coverage metrics and [6] with the increase number of test cases, wherein HTs are implemented by code model one and two; (c) presents the coverage of each method with the increase number of test cases.

trigger condition, the traditional verification detects fewer and fewer HTs. This is because the HT is likely to keep dormant during the verification if it is controlled by a rare trigger condition. On the contrary, UCI techniques detects more and more HTs with the decrease of the probability of the trigger condition. Since the probability of sub-trigger condition is reduced with the decrease of the probability of trigger condition, any sub-trigger conditions un-activated would cause the whole HT to be detected. Under our experimental environment, when the probability of trigger condition is set to be 1e-16 per clock cycle, HTs designed by proposed method can evade all detection methods.

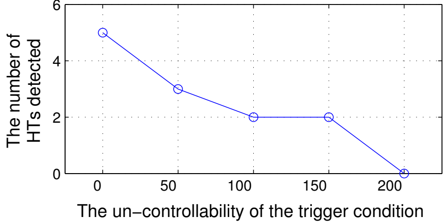


Fig. 6: The number of HTs detected under the constraint of the diverse un-controllabilities of the trigger condition

Finally, we discuss the impact of the controllability of the HT on its stealthiness. As shown in Fig. 6, the number of HTs detected decreases with the increase of the un-controllability of the trigger condition. This is because the requirement of the HT controllability could sacrifice some choices of trigger inputs, possibly resulting in poor trigger design. How to balance the controllability and stealthiness of HT mainly depends on the objective of the attacker.

V. Conclusion

In this paper, we propose a systematic hardware Trojan design and implementation methodology. Our approach designs and implements HTs that are not only hard to trigger but also easy to evade existing detection techniques based on UCI techniques. In addition, it considers the HT controllability in order to provide flexibility for attackers to control the HT. Experimental results demonstrate that the HTs designed with the proposed method can evade all existing HT detection techniques.

VI. Acknowledgement

This work was supported in part by a CUHK Direct Grant No. 2050488.

References

[1] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, pp. 10–25, 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] Inquiry into counterfeit electronic parts in the department of defense supply chain. *Committee Armed Services, United States Senate*, May 2012. http://www.levin.senate.gov/download/?id=24b3f08d-02a3-42d0-bc75-5f673f3a8c93.

[4] Defense science board task force on high performance microchip supply. *http://www.acq.osd.mil/dsb/reports/ADA435563.pdf*, 2005.

[5] M. Beaumont, B. Hopkins, and T. Newby. Hardware Trojans - prevention, detection, countermeasures (a literature review). Technical report, DTIC Document, 2011.

[6] M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, and J.M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Proc. IEEE Symposium on Security and Privacy*, pp. 159–172, 2010.

[7] 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.

[8] 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*, pp. 1–8, 2008.

[9] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno. A case study in hardware trojan design and implementation. *Information Security*, pp. 1–14, 2011.

[10] 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.

[11] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *Proc. IEEE Symposium on Security and Privacy*, pp. 296–310, 2007.

[12] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Proc. International Conference on Cryptographic Hardware and Embedded Systems*, pp. 382–395, 2009.

[13] Trust-hub website. *http://www.trust-hub.org*.

[14] Opencores website. *http://opencores.org/*.