Chapter 11
Hardware Trust Verification

Qiang Xu and Lingxiao Wei

11.1 Introduction

Hardware Trojans (HTs) are malicious alterations of normal integrated circuits (IC) designs. They have been a serious concern due to the ever-increasing hardware complexity and the large number of third-parties involved in the design and fabrication process of ICs.

For example, a hardware backdoor can be introduced into the design by simply writing a few lines of hardware description language (HDL) codes [1, 2], which leads to functional deviation from design specification and/or sensitive information leakages. Skorobogatov and Woods [3] found a "backdoor" in a military-grade FPGA device, which could be exploited by attackers to extract all the configuration data from the chip and access/modify sensitive information. Liu et al. [4] demonstrated a silicon implementation of a wireless cryptographic chip with an embedded HT and showed it could leak secret keys. HTs thus pose a serious threat to the security of computing systems and have called upon the attention of several government agencies [5, 6].

HTs can be inserted in ICs in almost any stage, e.g., specification, register-transfer level (RTL) design, logic synthesis, intellectual property (IP) core integration, physical design, and manufacturing process. Generally speaking, the likelihood of HTs being inserted at design time is usually much higher than that being inserted at manufacturing stage, because adversaries do not need to access foundry facilities to implement HTs and it is also more flexible for them to implement various malicious functions. Hardware designs can be covertly compromised by HTs inserted into the RTL code or netlist. These HTs may be implemented by rogue

Q. Xu (✉) • L. Wei
CUhk REliable computing laboratory (CURE Lab.), Department of Computer Science & Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong
e-mail: qxu@cse.cuhk.edu.hk; lxwei@cse.cuhk.edu.hk

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,
DOI 10.1007/978-3-319-49025-0_11

designers in the development team or integrated from a malicious third-party IP core. As it is not feasible to verify trustworthiness of ICs at fabrication stage, it is important to ensure that the design is HT-free before it enters the next phase of IC design cycle.

11.2 HT Classification

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 verification, stealthy HTs usually employ certain trigger condition that is controlled by dedicated trigger inputs and difficult to be activated with verification test cases.

The HTs inserted at design stage can be categorized according to their impact on the normal functionalities of original circuits. Trojans can either directly modify the normal functions or insert extra logic to introduce additional malicious behavior while maintaining original functionality.

They are named *bug-based HT* and *parasite-based HT*, respectively.

11.2.1 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. 11.1a 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. 11.1b. 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. 11.2a, 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. 11.1b, 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., [1, 2, 7–11]) belong to the parasite-based type, as discussed in the following.

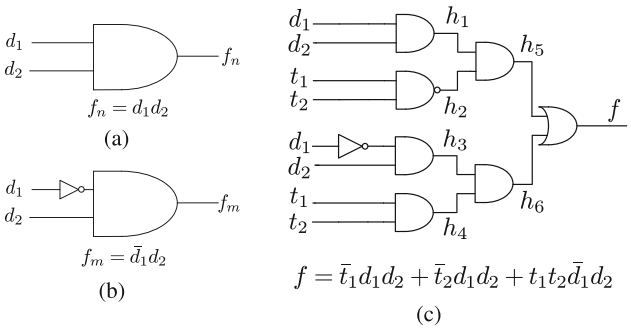

Fig. 11.1 HT classification with a simple example. (**a**) Original circuit. (**b**) Bug-based HT. (**c**) Parasite-based HT

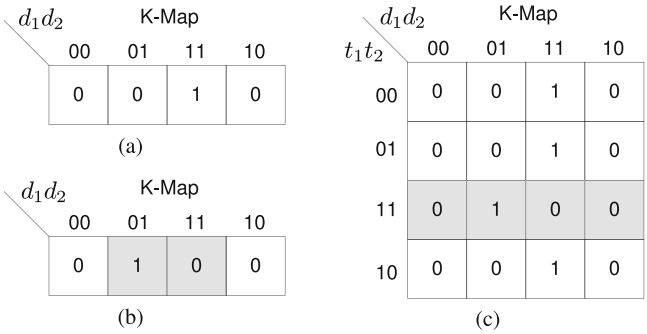

Fig. 11.2 (**a**) K-Map of original circuit in Fig. 11.1a; (**b**) K-Map of bug-based HT in Fig. 11.1b; (**c**) K-Map of parasite-based HT in Fig. 11.1c

11.2.2 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. Usually in order to escape the functional verification, trigger inputs are carefully selected and the trigger condition is designed to be an extremely rare event occurred with verification tests.

Let us examine the K-Map of the parasite-based HT-inserted circuit, as shown in Fig. 11.2c. 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. 11.2a), 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.

11.3 Verification Techniques for Hardware Trust

11.3.1 Functional Verification

Ideally, a HT can be detected by activating it and observing its malicious behaviors. During functional verification, a set of test cases would be applied to verify the functional correctness of the targeted design. As bug-based HTs utilize some functional inputs as triggers, it is highly likely to be activated and detected by extensive simulation in a functional verification process. Hence it is not good choice for adversaries. For parasite-based HTs, in theory, it can also be activated by enumerating all possible system state in an exhaust simulation. In reality, however, even with the sheer volume of states that exist in a simple design, functional verification can only cover a small subset of the functional space of a hardware design. Considering the fact that attackers have full controllability for the location and the trigger condition of their HT designs at design time, which are secrets to functional verification engineers, it is usually very difficult, if not impossible, to directly activate a parasite-based HT.

11.3.2 Formal Verification

Theoretically speaking, whether a hardware design contains HTs or not can be theoretically proven when a given trustworthy high-level system model (e.g., [12]) with formal verification. In practice, however, full formal verification of large circuits is still computationally infeasible. In addition, the golden model itself may not be available. Consequently, it is not appropriate to adopt formal verification to verify hardware trust and dedicated trust verification techniques are developed for parasite-based HT detection.

11.3.3 Trust Verification

Trust verification techniques flag suspicious circuitries based on the observation that HTs are nearly always dormant (by design) in order to pass functional verification. Such analysis can be conducted in a *dynamic* manner by analyzing which part of the circuit is not sensitized during functional verification, as in *UCI* [10] and VeriTrust [13]. Alternatively, *static* Boolean functional analysis can be used to identify suspicious signals with *weakly affecting* inputs, as in FANCI [14].

11.3.3.1 Unused Circuit Identification

Hicks et al. [10] first addressed the problem of identifying HTs inserted at design time, by formulating it as an unused circuit identification problem. The primary assumption in UCI is the hardware designs usually are verified with an extensive test suite while the HTs inside can successfully avoid being activated to manifest certain malicious behaviors on the outputs. Hence the "unused circuits," which are included in the design but do not affect any output during verification, are considered to be HT candidates.

One way to define unused circuit is based on the code coverage metrics used in verification (e.g., line coverage and branch coverage) such that those uncovered circuitries are flagged as suspicious malicious logic. However, it is not hard for attackers to craft circuits that are covered by verification but the trojan is never triggered. Suppose a HT is inserted into a HDL implementation as follows:

```
X <= (ctrl[0] == '0')?normal_func:trojan_func;
Y <= (ctrl[1] == '0')?normal_func:trojan_func;
Out <= (ctrl[0] == '0')?X:Y;
```

If the control values 00, 01, and 10 are included in the verification process, all three lines in the above code will be covered, but the out signal will always equal to the normal function. This simple definition based on coverage is easily defeated thus not appropriate for HT detection.

They defined "unused circuits" as follows. Consider a signal pair (s, t), where t is dependent on s. If $t = s$ throughout the entire functional verification procedure, the intermediate circuit between s and t is regarded as "unused circuit."

The UCI algorithm is performed in two steps. First, a data-flow graph is generated in which nodes are signals or state elements and edges indicate data flow between nodes. Based on the data-flow graph, a list of signal pairs or (data-flow pairs), where data flows from a source signal to a sink signal, are generated. The list includes both direct dependencies and indirect dependencies. Second, the HDL code is simulated with verification tests to detect the signal pairs that do not have data flows. On each simulation step, UCI checks inequality for each remaining data-flow pairs. If inequality is detected, the signal pair is regarded safe and are removed from suspicious list. Finally, after the simulation completes, a set of remaining data-flow pairs is identified in which the logic between the nodes does not affect the signal value from source to sink.

For the example circuit shown in Fig. 11.3a, a HT is inserted with malicious function $f_m = t_1 t_2 d_2$ with trigger condition $\{t_1, t_2\} = \{1, 1\}$. The data-flow graph of this example circuit is shown in Fig. 11.3b. From the data-flow graph, a list of 11 signal pairs can be constructed in the left part of Table 11.1.

Simulation is performed by applying test cases except those containing $\{t_1, t_2\} = \{1, 1\}$, which would activate the HT. Every signal pair's equality is recorded during simulation and their results are shown in Table 11.1. The signal pair (h_2, f) is always

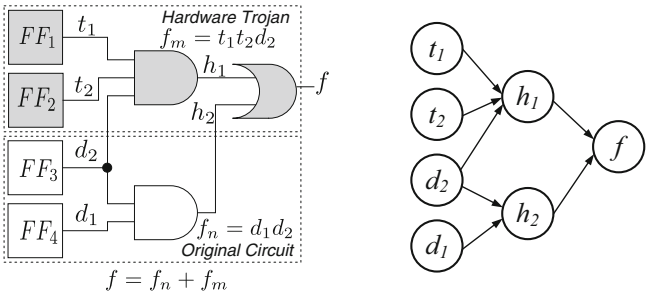

Fig. 11.3 (**a**) A HT-infected circuit with trigger inputs t_1 and t_2; (**b**) Data-flow graph of the HT-infected circuit

Table 11.1 Signal pairs generated from UCI

Signal pairs		Always equal under
Source	Sink	non-trigger condition
t_1	h_1	No
t_2	h_1	No
d_1	h_1	No
d_1	h_2	No
d_2	h_2	No
h_1	f	No
h_2	f	Yes
t_1	f	No
t_2	f	No
t_1	f	No
t_2	f	No

equal under non-trigger condition as h_1 is always 0. Hence the OR gate between them is regarded as "unused circuit" and is suspicious of HT infected.

With the above shown algorithm, given a specific hardware design in HDL code, the UCI algorithm in [10] traces all signal pairs during verification, and reports those ones for which the property $s = t$ holds throughout all test cases as suspicious circuitries.

One of the main limitations of UCI techniques is that they are sensitive to the implementation style of HTs. Later, Sect. 11.4.1.1 presented how to exploit this weakness to defeat UCI detection algorithms.

11.3.3.2 VeriTrust

In order to conquer the limitations of UCI, Zhang et al. proposed a HT detection method called *VeriTrust*. VeriTrust [13] flags suspicious circuitries by identifying potential trigger inputs used in HTs, based on the observation that these inputs

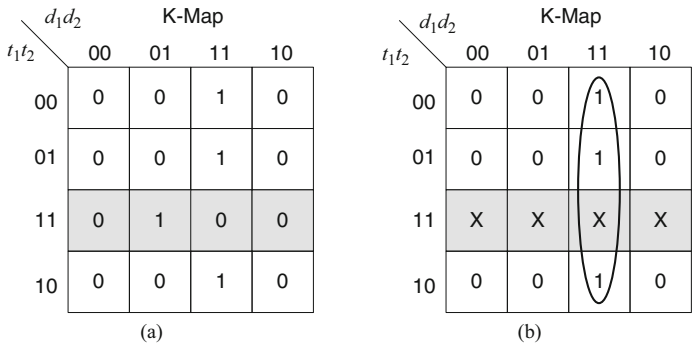

Fig. 11.4 (**a**) A HT-infected circuit with trigger inputs t_1 and t_2; (**b**) Data-flow graph of the HT-infected circuit

keep dormant under non-trigger condition (otherwise HTs would have manifested themselves) and hence are **redundant** to the normal logic function of the circuit.

The intuition behind *VeriTrust* is the fact that any HT-infected signal must be driven by at least one dedicated trigger input. Thus the function of a HT-infected signal can be represented as $f = C_n f_n + C_m f_m$, wherein f_n and f_m denote the circuit's normal function and HT's malicious function while C_n is non-triggering condition and C_m the triggering condition, respectively. As C_m never appear in the verification process, all its entries in K-map can be set to don't-cares without affecting the normal function. After logic simplification, the function $f = f_n$, which means the triggering inputs, becomes redundant.

Here is an illustration based on the example circuit from Fig. 11.1c. Its Boolean function is $f = \bar{t}_1 d_1 d_2 + \bar{t}_2 d_1 d_2 + t_1 t_2 \bar{d}_1 d_2$. The K-map of this function is shown in Fig. 11.4a. Suppose the verification has verified every entries in K-map except those with triggering condition, highlighted with gray color in Fig. 11.4a. These entries would be set to don't-cares in *VeriTrust*, as shown in Fig. 11.4b. By logic simplification, the original Boolean function reduced to the normal function $d_1 d_2$, leaving the triggering input t_1, t_2 redundant.

VeriTrust can be considered as an "unused input identification" technique by looking for redundant inputs after setting all un-activated entries in verification tests to be don't-cares.

The overall framework of *VeriTrust* is shown in Fig. 11.5 which contains two parts: *tracer* and *checker*. The tracer traces verification tests to identify those signals that contain un-activated entries by tracing mechanisms. Then the checker analyzes these signals and determines whether any of them indeed contain redundant inputs and hence are potentially affected by HTs.

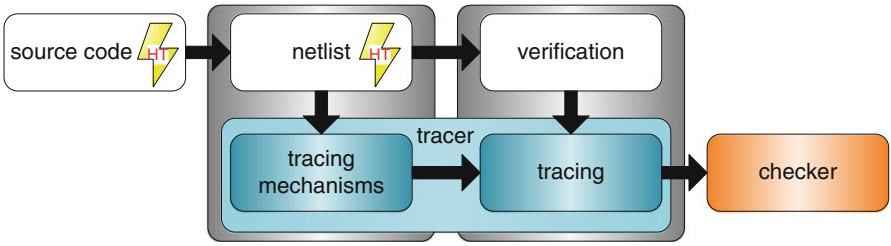

Fig. 11.5 The overview of *VeriTrust*

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, a much more compressed tracing mechanism is proposed.

Before discussing the details, it is necessary to 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, Zhang et al. introduced three new terms, *malicious on-set minterm*, *malicious off-set maxterm*, and *dummy term* that compose malicious function as follows.

Definition 1. 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 2. 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 3. 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 identify dedicated trigger inputs.* Figure 11.6 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

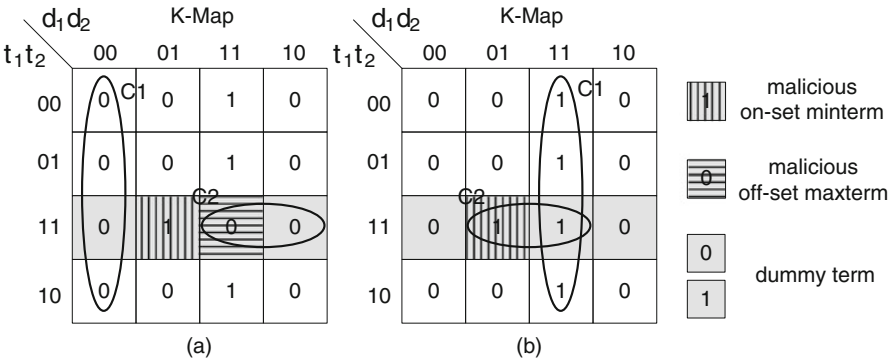

Fig. 11.6 Two HT-affected circuits triggered by $\{t_1, t_2\} = \{1, 1\}$

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), following observations have been obtained:

- 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 minterms (maxterms) of malicious on-set minterms (malicious off-set maxterms) in the normal function are off-set (on-set). For example, in Fig. 11.6a, one malicious off-set maxterm is combined with one dummy term (circled by C2), and in Fig. 11.6b, 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. 11.6a, b shows that the dummy term is combined with terms in the normal function; the circle C2 in Fig. 11.6a, 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.

The tracing procedure might still be time-consuming if the number of products and sums to be traced is large. Tracing overhead can be further reduced by periodically removing those activated sums and products and those un-activated ones whose signal is determined to be HT-free with the 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.

Checker

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, three optimization methods are adopted in sequence 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. 11.6a 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. 11.6b 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, checker 1, checker 2, and checker 3 are run in a consecutive manner. For each signal examined, if a more efficient checker (e.g., checker 1) finds out a redundant input for a particular signal with un-activated products/sums, the products/sums are marked as a suspicious HT-affected signal. Otherwise, next checker is used 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.

Algorithm 11.1: Suspicious wires detection in FANCI

```
for all modules m do
    for all gates g in m do
        for all output wires w of g do
            T ← TruthTable(FanInCone(w));
            V ← Empty vector of control values;
            for all columns c in T do
                Compute control value of c;
                Add control(c) to vector V;
            end for
            Compute heuristics for V;
            Decide w as suspicious or not;
        end for
    end for
end for
```

11.3.3.3 FANCI

FANCI [14], which stands for Functional Analysis for Nearly unused Circuit Identification, is a kind of *static* Boolean function analysis to find signals with weakly affecting inputs. It is based on the observation that a HT trigger input generally has a *weak* impact on output signals. Thus these weakly affecting inputs are probably inserted by adversaries as backdoor triggers. The primary goal of FANCI is to identify the "weakly affecting" dependency between inputs and outputs via a metric called *control value*, which measure the degree of control that an input has on the outputs of a digital circuit.

The overall detection algorithm can be seen in Algorithm 11.1. FANCI examines outputs of every gate in all modules inside the hardware design and constructs functional truth table from its fan-in cone. For each input feeding into the examined output, FANCI determines whether the input is suspicious by *control value* computed from truth table. Given an input wire w_i and output wire w_o, FANCI algorithm traverses all rows in the truth table of w_o, denoted by T, and count rows in T that w_o changed when flipping the value of w_i. The *control value* of w_i on w_o is defined as

$$\mathrm{CV}(w_i, w_o) = \frac{count}{size(T)} \tag{11.1}$$

where *count* denotes the total number of rows the change of w_i affecting the value of w_o while $size(T)$ denotes the size of truth table (i.e., the total number of rows in T).

For example, shown in Fig. 11.3a, which function can be denoted as $f = d_1d_2 + t_1t_2d_2$. Its truth table is shown in Table 11.2. There are only two rows, highlighted in gray, under which flipping t_1 leads to the change of the output, and hence the *control value* of t_1 on f is $\mathrm{CV}(t_1, f) = 2/2^4 = 0.125$. In the similar manner, a vector of *control values* $\mathbf{V}$ is acquired for f as $\mathbf{V} = [0.125, 0.125, 0.375, 0.625]$ containing the control values for input t_1, t_2, d_1, d_2. Since the sizes of truth tables

Table 11.2 Truth table of circuit $f = d_1d_2 + t_1t_2d_2$

t_1	t_2	d_1	d_2	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

grow exponentially with respect to the number of input wires, computing *control values* deterministically is exponentially hard. Necessary optimization is required for efficient and practical detection. They proposed to compute approximate *control value* by selecting a subset of rows from the complete truth table at uniformly random. For instance, N cases are chosen randomly for computing and for each case w_i is flipped and the total number of times that w_o changes is recorded, denoted by n. The approximate *control value* is $\frac{n}{N}$. It shall be noted that rows in truth table are selected uniformly at **random** to prevent potential adversaries from exploiting the sampling procedure and designing HTs evade FANCI.

When the vector of control values is computed for a signal under analysis, it is not trivial to determine whether it contains a HT trigger. Having only one weakly affecting input or a input that is only borderline weakly affecting is not sufficiently suspicious as it could simply be an inefficient implementation. FANCI includes the several heuristics, such as median and mean, to decide whether the signal is suspicious by considering all elements in the CV vector.

Median If an output is affected by HT triggers, the median is often close to zero. If the distribution in CV vector is irregular, the median may bring in unnecessary false positives.

Mean This metric is similar to median, but slightly sensitive to outliers. It is more effective when there are few unaffecting dependencies.

Median and Mean Consider the signals with both extreme median and mean values to mitigate the limitations and diminish false positives.

Triviality This metric calculates a weighted average of the vector of control values. Each weight of the wires in CV vector is measured by how often they are the only wire influencing the output to determine how much an output is influenced overall by its input. Suppose an output wire with $w_o = w_{i_1} \oplus \ldots \oplus w_{i_n}$ is analyzed by FANCI, all input wires have the control value of 1.0, but it is never the case that one input completely controls the output. Thus the triviality, i.e., the weight of every input is 0.5.

Table 11.3 HT detection with hardware functional verification and trust verification

	Static/dynamic	Detection method	Runtime
Functional verification	Dynamic	Activate the HT	Good
UCI by code coverage	Dynamic	Identify uncovered parts	Good
UCI by Hicks et al. [10]	Dynamic	Identify equal signal pair	Fair
VeriTrust [13]	Dynamic	Identify HT trigger inputs	Fair
FANCI [14]	Static	Identify weakly affecting inputs	Fair

	False negatives	False positive
Functional verification	HTs with rare trigger condition	None
UCI by code coverage	HTs in [2]	Few with thorough verification
UCI by Hicks et al. [10]	HTs in [2, 11]	Some with thorough verification
VeriTrust [13]	Unknown	Some with thorough verification
FANCI [14]	Possible with low threshold	Many with high threshold

For each metric, it is necessary to have a cut-off threshold for what is suspicious and what is not. The value is chosen either a priori or after examining the distribution of computed values.

11.3.3.4 Discussion

Table 11.3 summarizes the characteristics of existing solutions for HT detection. Since dynamic trust verification techniques (i.e., UCI and VeriTrust) analyze the corner cases of functional verification for HT detection, these two types of verification techniques somehow complement each other. Generally speaking, with more FV tests applied, the possibility for HTs being activated is higher while the number of suspicious circuitries reported by UCI and *VeriTrust* would decrease. As a static solution that does not depend on verification, one unique advantage of FANCI over the other solutions is that it does not require a trustworthy verification team. On the other hand, however, adversaries could also validate their HT designs using FANCI without necessarily speculating the unknown test cases used to catch them.

All trust verification techniques try to eliminate *false negatives* (a false negative would mean a HT that is not detected) whilst keeping the number of *false positives* as few as possible in order not to waste too much effort on examining benign circuitries that are deemed as suspicious. However, their detection capability is related to some user-specified parameters and inputs during trust verification. For example, FANCI defines a *cut-off threshold* for what is suspicious and what is not during Boolean functional analysis. If this value is set to be quite large, it is likely to catch HT-related wires together with a large number of benign wires. This, however, is a serious burden for security engineers because they have to evaluate all suspicious wires by code inspection and/or extensive simulations. If this value is set to be quite small, on the contrary, it is likely to miss some HT-related wires. Similarly, if only a

small number of FV tests are applied, UCI and *VeriTrust* would flag a large number of suspicious wires (all wires in the extreme case when no FV tests are applied), which may contain HT-related signals but the large amount of false positives make the following examination procedure infeasible.

11.4 Stealthy HT Designs Defeating Trust Verification

HT design and HT identification techniques are like arms race, wherein designers update security measures to protect their system while attackers respond with more tricky HTs. With the state-of-the-art hardware trust verification techniques such as UCI, *VeriTrust*, and FANCI being able to effectively identify existing HTs, no doubt to say, adversaries would adjust their tactics of attacks accordingly and *it has been revealed that new types of HTs can be designed to defeat these hardware trust verification techniques.*

11.4.1 HTs Evade UCI

UCI tracks signal pairs whose values remain the same during the verification process. The circuits between these signals are regarded as "unused circuit" and are potential HT candidates. The obvious way to evade UCI is to make all dependable signal pairs in HT differ at least once under non-trigger condition.

11.4.1.1 Motivational Case

Sturton et al. [11] showed a simple design consisting only of two multiplexers evades UCI. The design is illustrated in Fig. 11.7. This circuit evades detection by UCI because there is no dependent signal pairs that are always equal under non-trigger condition. There are two non-trigger inputs d_1, d_2 and two trigger inputs t_1, t_2, with trigger condition $\{t_1, t_2\} = \{1, 1\}$. The output function $f = d_0 d_1 + t_0 t_1 \bar{d}_1$ and its truth table are shown in Table 11.4. Under trigger condition, the malicious function $f = d_0 \bar{d}_1$ is showed while under non-trigger condition, the normal function $f = d_0 d_1$ is facilitated.

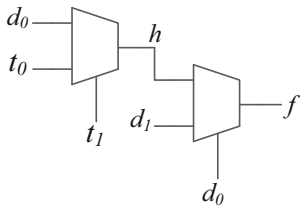

Fig. 11.7 Example defeating UCI

Table 11.4 Truth table of circuits in Fig. 11.7

	t_0	t_1	d_0	d_1	h	f	Comments
1	0	0	0	0	0	0	
2	0	0	0	1	0	0	$f \neq d_1$
3	0	0	1	0	1	0	$h \neq t_0, t_1$
4	0	0	1	1	1	1	$f \neq t_0, t_1$
5	0	1	0	0	0	0	
6	0	1	0	1	0	0	
7	0	1	1	0	0	0	$h \neq d_0$
8	0	1	1	1	0	1	$f \neq h$
9	1	0	0	0	0	0	
10	1	0	0	1	0	0	
11	1	0	1	0	1	0	$f \neq d_0$
12	1	0	1	1	1	1	
13	1	1	0	0	1	1	Trigger
14	1	1	0	1	1	1	Trigger
15	1	1	1	0	1	0	Trigger
16	1	1	1	1	1	1	Trigger

In Table 11.4, the rows in gray are the signal values under triggering condition, thus it cannot be seen during the verification tests. The final column notes when an input makes a dependent pair of signals unequal. If the verification test includes non-triggering patterns in row 2, 3, 4, 7, 8, 11, UCI would not flag any part of the circuit as malicious. Since the trigger condition for all of these inputs is false and the normal function is $d_0 d_1$, this circuits behaves identically with an AND gate in all test cases.

A rogue designer may utilize this fact and replace any AND gate with the malicious circuit in Fig. 11.7 and can hide from UCI detection. If this affects some security critical element of hardware, a hidden vulnerability is introduced by this HT.

The reason why UCI missed this behavior is that there is no intermediate function equivalent to the output function so that UCI cannot find a short-circuit with the output. The key idea to defeat UCI is to create a non-trigger function that is *not* equivalent to any internal functions.

11.4.2 HT Design Against UCI

Sturton et al. proposed to construct malicious Trojan in an incremental manner. They first built a set of small circuits that evade UCI and construct more stealthy HTs by combing these small circuits with basic circuit-building elements (e.g., AND gate, OR gate, MUX, etc.) via exhaustive search. Every newly created circuit is verified by UCI to determine whether it can evade UCI.

The intuition behind this incremental search is based on the fact every subcircuit of a UCI-evading circuit also evades UCI. This is simple to understand by considering the definition of "unused circuit." If a circuits evades UCI, meaning that every internal signal inside this circuits does not always equal to the signals in its fan-in cone. Thus the circuits between each internal signal and its primary inputs are a subcircuit and it evades UCI.

Before presenting the HT design algorithm, Sturton et al. defined several concepts for the ease of discussion:

Definition 4. An *admissible circuit* is a circuit that (1) contains exactly one trigger condition, (2) has at least one trigger inputs, (3) its output is independent from trigger input under non-trigger condition.

Definition 5. A circuit is *obviously malicious* if there exist two valuations to the inputs of the circuit, $x = (d, t)$, $x' = (d, t')$, in which t is a non-trigger condition while t' is trigger condition and the output of the circuit is different under these two conditions.

Definition 6. A circuit is a *stealthy circuit* if it evades UCI, i.e., there is no pair of dependent signals always equal in verification tests.

These three definitions perfectly characterize the three aspects of HTs that evades UCI. Firstly, HT shall contain a malicious function that can be triggered in specific conditions, i.e., obviously malicious. Second, HT shall pass the design-time verification, the output has nothing to do with trigger inputs, i.e., admissible. Finally, it cannot contain any signals that would be flagged as suspicious by UCI. The goal of HT construction algorithm is to find the class of HTs that contains above mentioned characters.

Algorithm 11.2 shows the proposed way to generate HTs from scratch. HT designers first shall fix the number of input signals to the circuit and the size of the circuit. Also, the basic blocks for building the circuit shall be determined. The construction algorithm maintains a workqueue of newly formed circuits which would serve as building blocks for future circuits. The workqueue initially contains the circuits that contains only one input.

Roughly speaking, in each iteration of the algorithm, one circuit is removed out of the workqueue and consider all ways to expand it to make a new stealthy circuit. If a new circuit is found satisfying three definitions, it is added to the database of all constructed HTs. The algorithm can be run multiple times with different parameters (e.g., number of inputs, size of circuit, basic building gates, etc.), and more versatile HTs would be added to the database.

After the searching algorithm is finished, a database containing circuits with various normal functions and malicious functions is formed. Rogue designers can replace arbitrary circuit in the normal design with HT-infected circuit with the same normal function by searching the database.

Algorithm 11.2: Construct circuits to defeat UCI

```
   // Initial Circuit, one for each input
 1 C_0 = (d_0);C_1 = (d_1);...;C_n = (t_m);
   // Set of circuits found that evades UCI
 2 completed_circuits = ∅;
   // Set of circuits used as building block for larger circuits
 3 workqueue = {C_0, C_1, ..., C_n};
   // Set of basic gates in building circuits
 4 gate_basis = {AND, OR, NOT, NAND, 2-input MUX};
 5 while length(workqueue) > 0 do
 6     curr_circuit = workqueue.pop();
 7     if curr_circuit is stealthy, admissible and oblivious malicious then
 8         Print curr_circuit;
 9     else
10         for all gate in gate_basis do
11             for all circ in completed_circuits ∪ {curr_circuit} do
12                 new_circuit = gate(curr_circuit, circ);
13                 if new_circuit is stealthy and not in completed_circuits then
14                     workqueue.append(new_circuit);
15                 end if
16             end for
17         end for
18         completed_circuits.add(curr_circuit);
19     end if
20 end while
```

11.4.3 HTs Evade VeriTrust

As discussed earlier, *VeriTrust* flags suspicious HT trigger inputs by identifying those inputs that are redundant under verification. Consequently, as it is shown in [15], the key idea to defeat *VeriTrust* is to make HT-affected signals driven by non-redundant inputs only under non-trigger condition.

11.4.3.1 Motivational Case

For any input that is not redundant under non-trigger condition, the following lemma must be true.

Lemma 1. *Consider a HT-affected signal whose Boolean function is* $f(h_1, h_2, \ldots, h_k)$. *Any input,* h_i, *is not redundant under non-trigger condition, as long as the normal function, denoted by* f_n, *cannot be completely represented without* h_i.

Proof. Since f_n cannot be completely represented without h_i, there must exist at least one pattern for all inputs except h_i under which $f_n(h_i = 0) \neq f_n(h_i = 1)$. Therefore, h_i is not redundant.

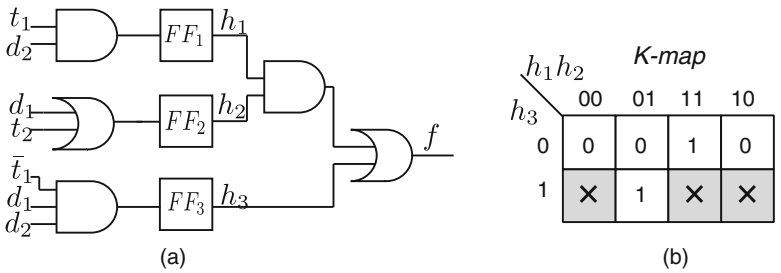

Fig. 11.8 Motivational example for defeating VeriTrust (**a**) HT-infected circuit (**b**) K-map

Inspired by Lemma 1, Fig. 11.8a shows a HT-infected circuit that is revised according to the circuit shown in Fig. 11.4, wherein the HT is activated when $\{t_1, t_2\} = \{1, 1\}$. In this implementation, the malicious product $t_1t_2d_2$ is combined with the product $t_1d_1d_2$ from the normal function and hidden in the fan-in cones of h_1 and h_2, where $h_1 = t_1d_2$ and $h_2 = d_1 + t_2$. The K-map of f is shown in Fig. 11.8b, where entries that cannot be activated under non-trigger condition are marked as "don't-cares." For this circuit, VeriTrust, focusing on the combinational logic, would verify four signals, f, h_1, h_2, and h_3. According to the K-map shown in Fig. 11.8b, it is clear that h_1, h_2, and h_3 are not redundant for f under non-trigger condition. Moreover, h_1, h_2, and h_3, which are not HT-affected signals, have no redundant inputs as well, since all of their input patterns can be activated under non-trigger condition. Therefore, the HT in Fig. 11.8a is able to evade VeriTrust.

By examining the implementation of this motivational case, it is clear that the mixed design of the trigger and the original circuit makes trigger condition for the HT-affected signal not visible for VeriTrust. In order to differentiate existing HTs and HTs like the one shown in Fig. 11.8a, two new terms are introduced in [15]: the *explicitly triggered HT* and the *implicitly triggered HT* as follows.

Definition 7. A HT is **explicitly triggered** if in the HT-affected signal's fan-in logic cone, there exists an input pattern that uniquely represents the trigger condition.

Definition 8. A HT is **implicitly triggered** if in the HT-affected signal's fan-in logic cone, there does not exist any input pattern that uniquely represents the trigger condition.

All explicitly-triggered HTs can be detected by VeriTrust, as they contain dedicated trigger inputs which can be identified by VeriTrust. The HT shown in Fig. 11.8a is implicitly-triggered, since the trigger condition is hidden in the h_1h_2 which also contains certain circuit's normal functionalities.

The HT design method shown in next section is motivated by the above example and observations by implementing implicitly-triggered HTs to defeat VeriTrust.

11.4.3.2 HT Design Against VeriTrust

As *VeriTrust* only focused on detecting HT trigger inputs in the combinational logic block wherein the output is HT-affected signal. The HT design methodology evading *VeriTrust* also focused on the combinational logic. According to Lemma 1, the approach to defeat *VeriTrust* implements the implicitly triggered HT with the following two steps:

- Combine all malicious on-set terms[1] with on-set terms from normal function, and re-allocate sequential elements (e.g., flip-flops) to hide the trigger in multiple combinational logic blocks.
- Simplify all remaining on-set terms and re-allocate sequential elements if they contain trigger inputs.

Note that, only on-set terms are selected, since the circuit can be explicitly represented by the sum of all on-set terms.

The defeating method against *VeriTrust* can be further illustrated as follows. Consider a circuit with an explicitly triggered HT, and its Boolean function can be represented by

$$f = \sum_{c_{n_i} \forall C_n,\, p_{n_j} \forall P_n,} c_{n_i} p_{n_j} + \sum_{c_{m_i} \forall C_m,\, p_{m_j} \forall P_m} c_{m_i} p_{m_j}, \tag{11.2}$$

where P_n and P_m driven by functional inputs denote the set of all patterns that make the normal function and malicious function output logic "1", while C_n and C_m driven by trigger inputs denote the set of non-trigger conditions and the trigger conditions, respectively.

For the sake of simplicity, the analysis began with the case where the malicious function contains only one malicious on-set term, $c_{m_0} p_{m_0}$. Suppose $c_{n_0} p_{n_0}$ from the normal function is selected to combine with $c_{m_0} p_{m_0}$. Let f'_n be all the on-set terms from the normal function except $c_{n_0} p_{n_0}$. Then, f can be given by

$$f = f'_n + (c_{n_0} p_{n_0} + c_{m_0} p_{m_0}). \tag{11.3}$$

Suppose $c_{n_0} p_{n_0}$ and $c_{m_0} p_{m_0}$ have the common literals, $c^c p^c$, and then

$$f = f'_n + c^c p^c (c^r_{n_0} p^r_{n_0} + c^r_{m_0} p^r_{m_0}), \tag{11.4}$$

where

$$\begin{aligned} c_{n_0} &= c^c c^r_{n_0}; \quad & p_{n_0} &= p^c p^r_{n_0}; \\ c_{m_0} &= c^c c^r_{m_0}; \quad & p_{m_0} &= p^c p^r_{m_0}. \end{aligned} \tag{11.5}$$

[1]Malicious on-set term is the on-set term in the malicious function whose adjacent terms in the normal function are off-set [13]. On-set term and off-set term are terms that make the function output logic "1" and logic "0", respectively.

After that, the circuit is re-synthesized and the flip-flops are re-allocated to make f become

$$f = h_1 h_2 + h_3, \tag{11.6}$$

where

$$\begin{aligned} h_1 &= c^c p^c \\ h_2 &= c^r_{n_0} p^r_{n_0} + c^r_{m_0} p^r_{m_0}. \\ h_3 &= f'_n \end{aligned} \tag{11.7}$$

h_1, h_2, and h_3 are outputs of the re-allocated flip-flops.

As can be observed in Eqs. (11.6) and (11.7), the key of the defeating method is to extract common literals from the malicious on-set term and the on-set term from the normal function and hide the trigger into different combinational logic. With the above, f, h_1, h_2, and h_3 would have no redundant inputs under non-trigger condition.

For multiple malicious on-set terms, the above method can also be used to combine each of them with one on-set term from the normal function and then hide the trigger in different combinational logic blocks. Finally, the output function f can be represented as:

$$f = \sum_{i=0}^{k-1} (h_{2i+1} h_{2i+2}) + h_{2k+1}, \tag{11.8}$$

where

$$\begin{aligned} h_{2i+1} &= c^{c_i} p^{c_i} \\ h_{2i+2} &= c^{r_i}_{n_i} p^{r_i}_{n_i} + c^{r_i}_{m_i} p^{r_i}_{m_i}. \\ h_{2k+1} &= f'_n \end{aligned} \tag{11.9}$$

It is easy to prove that $h_1, h_2, \ldots, h_{2k+1}$ and f have no redundant inputs under non-trigger condition. Note that the HT can be spread over multiple sequential levels by further combining the trigger logic driving $h_1, h_2, \ldots, h_{2k+1}$ with normal logic.

The defeating approach shown above can defeat *VeriTrust* in theory if a complete verification is performed. However, in practice, due to stringent time limit and ever-increasing circuit complexity, it is very hard to guarantee the verification is enough for HT detection. Hence probability of HT trigger inputs being flagged as redundant inputs still exists because of insufficient verification. The authors proposed the following three optimizations:

- Combine simplified malicious products (rather than malicious on-set terms) with on-set terms from the normal function, so that fewer terms from the normal function are required to be activated to evade VeriTrust.

Algorithm 11.3: The flow to defeat VeriTrust

```
1 Simplify Boolean function of this combinational logic;
2 Conduct the simulation with tests guessed by attackers to obtain the probability of each
  product;
3 foreach simplified malicious product do
4 |   Greedily combine it with the product from the normal function with the largest
  |   activation probability and hide the trigger in different combinational logic blocks;
5 end foreach
6 Re-allocate flip-flops for the remaining products.
```

- Choose simplified products from the normal function to be combined with simplified malicious products, so that any of the terms in the product from the normal function being activated can make HT evade VeriTrust.
- Choose those simplified products from the normal function with high activation probabilities to be combined with malicious products. This method requires the knowledge about the probability of products from the normal function, which can be estimated by speculating on the test cases used in functional verification [2, 11].

The flow to defeat *VeriTrust* is illustrated in Algorithm 11.3. It first simplifies the Boolean function of the combinational logic of HT-affected signals, and then conducts simulation with speculated test cases to obtain the probability of each product. After that, a loop is used to hide HT triggers whenever possible. In each iteration, one malicious product is combined with one product from the normal function with the largest activation probability and it is hidden in different combinational logic blocks. At last, flip-flops are re-allocated for the remaining products.

11.4.4 HTs Evade FANCI

Since FANCI identifies signals with weakly affecting inputs within the combinational logic block, the key idea to defeat FANCI is to make the control values of all HT-related signals comparable to those of functional signals.

11.4.4.1 Motivational Case

The authors in [15] started with the case shown in Fig. 11.9 to illustrate the key idea of defeating FANCI. Figure 11.9a, b presents a regular multiplexer (MUX) and a malicious one with a rare trigger condition, respectively. FANCI is able to differentiate the two types of MUXes and flag the malicious one since the trigger inputs, denoted by $t_0, t_1, \ldots, t_{63}$, have very small control values for the output $O(\frac{1}{2^{65}})$.

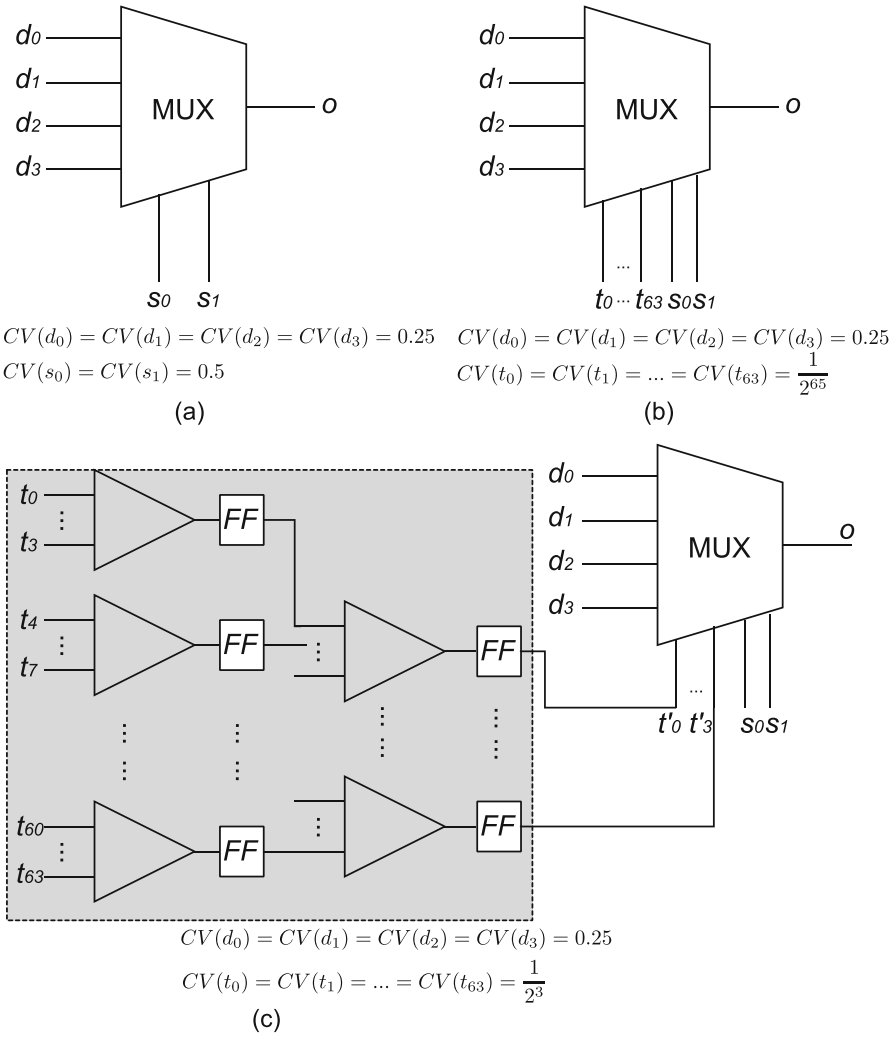

Fig. 11.9 Motivational example for defeating FANCI. (**a**) A standard MUX. (**b**) A malicious MUX. (**c**) A malicious MUX with modified implementation

From this example, we can see that the main reason for HT-related signals (e.g., *o* in Fig. 11.9b) having weakly affecting inputs is that it is driven by a number of trigger inputs in its fan-in combinational logic cone. Consider a signal driven by a combinational logic block with *m* trigger inputs and *n* functional inputs. The size of the truth table for this particular HT-related signal is given by:

$$size(T) = 2^{m+n}. \tag{11.10}$$

For any trigger input, denoted by t_i, those input patterns under which t_i influences the output should meet two requirements: (1) all trigger inputs other than t_i are driven by the trigger values[2]; (2) flipping t_i results in the change of the output value. There are in total 2^{n+1} input patterns meeting the first requirement. Among them, how many further satisfying the second requirement depends on the actual difference between the malicious function and the normal function, because they may output the same value under certain functional inputs. At the same time, they cannot always output the same value because otherwise there would be no malicious behavior. Therefore, the number of input patterns satisfying both requirements is bounded at:

$$2^1 \leq counter \leq 2^{n+1}. \tag{11.11}$$

With Eqs. 11.10 and 11.11, the control value of t_i on the corresponding HT-related signal is bounded at:

$$\frac{1}{2^{n+m-1}} \leq \mathrm{CV}(t_i) = \frac{counter}{size(T)} \leq \frac{1}{2^{m-1}}. \tag{11.12}$$

In order to make FANCI difficult to differentiate HT-related signals and function signals, the control values of HT-related signals shall be comparable to those of functional signals. As indicated by Eq. (11.12), reducing m has an exponential impact on the increase of control values. Thus, the authors modify the implementation of the malicious MUX by balancing these trigger inputs into multiple sequential levels (see Fig. 11.9c). In this way, the number of trigger inputs is controlled to be no more than four for any combinational block, rendering the control value of each trigger input comparable with those of functional inputs.

Motivated by the above, the approach of defeating FANCI is to reduce the number of trigger inputs in all the combinational logic blocks that drive HT-related signals, and it can be achieved by spreading HT trigger inputs among multiple sequential levels.

11.4.5 HT Design Against FANCI

In this section, two kinds of HTs are considered. Typically, the trigger of a stealthy HT consists of both combinational part (❶ in Fig. 11.10) and a sequential part (❷ in Fig. 11.10). The defeating algorithm for FANCI need to deal with them separately because different methods are adopted to handle the extra delay induced by additional sequential levels. Algorithm 11.4 presents the flow to defeat FANCI.

For the combinational logic blocks in ❶, the defeating method is similar to the one shown in Fig. 11.9c. As can be seen, the original trigger combinational logic

[2]Trigger values are logic values for trigger inputs to satisfy trigger condition.

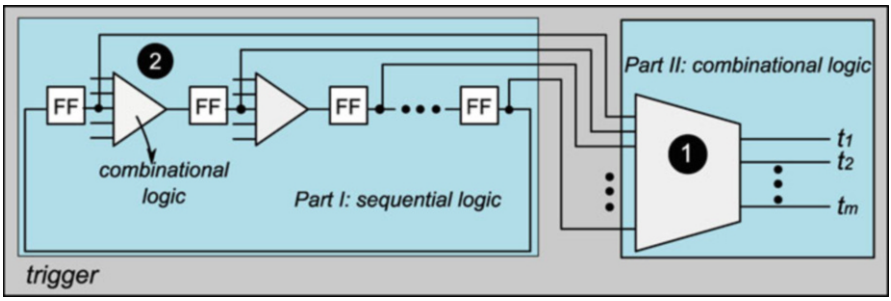

Fig. 11.10 Trigger design for a typical HT

Algorithm 11.4: The flow to defeat FANCI

```
1  N_T = 2;
2  do
3  |  DefeatFANCI(N_T++);
4  while (The hardware cost is larger than a given constraint.);
   /* One step to defeat FANCI                                   */
5  DefeatFANCI(N_T)
   |  /* For combinational logic of ❶                           */
6  |  foreach The fan-in cone of the input of the flip-flop do
7  |  |  if the number of trigger inputs > N_T then
8  |  |  |  Balance the trigger in the multiple sequential levels;
9  |  |  end if
10 |  end foreach
   |  /* For combinational logic of ❷                           */
11 |  Find out the maximum number of trigger signals, denoted by N_max, within a
   |  combinational logic cone;
12 |  if N_max > N_T then
13 |  |  Introduce multiple small FSMs until N_max ≤ N_T;
14 |  end if
15 end
```

with a large number of trigger inputs is spread among multiple combinational logic blocks, such that the number of trigger inputs in each combinational logic block is no more than N_T (a value used to tradeoff HT stealthiness and hardware overhead). As shown in Algorithm 11.4 (Lines 6–10), only the inputs of flip-flops are examined, rather than all signals. This is because, as long as the number of trigger inputs for the input of every flip-flop is smaller than N_T, the number of trigger inputs for every internal signal is also smaller than N_T.

The sequential part of a HT trigger can be represented by a finite-state machine (FSM) and the trigger inputs are used to control the state transitions (see Fig. 11.11a). For the combinational logic blocks in ❷ that sits inside the FSM, the above defeating method is not applicable because it introduced extra sequential levels. The additional pipeline delay incurred would change trigger condition.

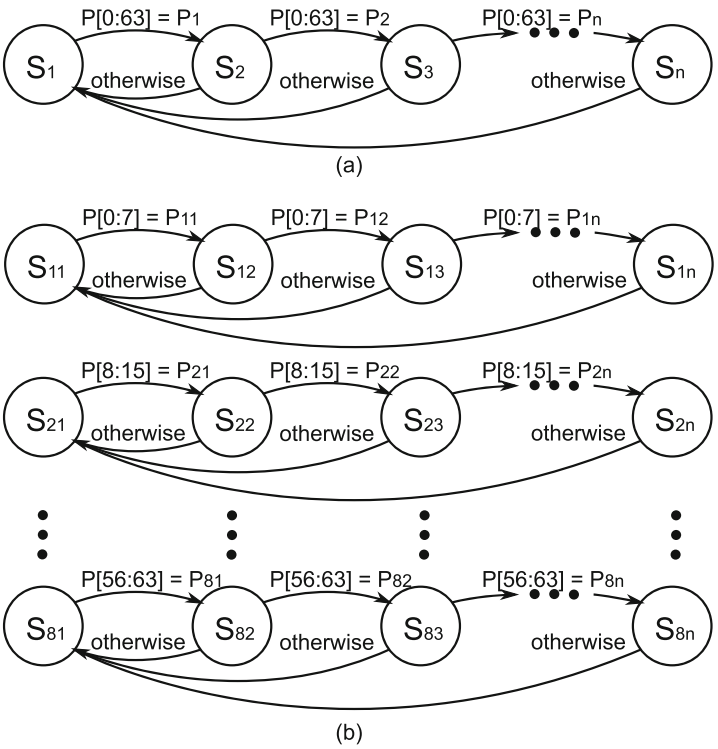

Fig. 11.11 The proposed sequential trigger design to defeat FANCI. (**a**) The original FSM. (**b**) The multiple small FSMs

Instead, the original FSM is partitioned into multiple small FSMs, e.g., as shown in Fig. 11.11b, the FSM with 64 trigger inputs is partitioned into eight small FSMs. By doing so, the number of trigger inputs in each small FSM is reduced to eight for this example, and it can be further reduced by introducing more FSMs. The HT is triggered when all the small FSMs reach certain states simultaneously.

Note that the proposed defeating method against FANCI has no impact on both circuit's normal functionalities and HT's malicious behavior, because this method only manipulates the HT trigger design, which is separated from the original circuit and the HT payload.

As the stealthiness of a HT is mainly determined by the number of trigger inputs in each combinational logic, this is achieved by finding the value of N_T in a greedy manner. That is, as shown in Algorithm 11.1, we start with $N_T = 2$ and gradually increase it until the cost of applying the defeating method is lower than the given constraint.

11.4.6 Discussion

The defeating approach against FANCI and that against *VeriTrust* do not interfere with each other. On the one hand, defeating algorithm for FANCI focuses on reducing the number of trigger inputs in the combinational logic blocks used in HT triggers without changing their logic functions; on the other hand, defeating method for *VeriTrust* implements the implicitly triggered HT without increasing the number of trigger inputs in any combinational logic.

Moreover, the HTs designed for defeating FANCI and *VeriTrust* would not influence the stealthiness of HT designs in Sect. 11.4.1.1 against UCI. Firstly, these methods do not change HT trigger condition and hence it has no impact on functional verification. For the UCI technique in [10], on the one hand, the signals introduced in algorithm for defeating FANCI are within the HT trigger unit driven by different trigger inputs and they are unlikely to be always equal during functional verification; on the other hand, *VeriTrust* defeating method combines parts of the normal functionalities with the HT trigger and hence is also unlikely to create equal signal pairs during verification.

With the above, all three HT design methodologies can be mixed in a single design to make the HT resistant to all known trust verification techniques while still passing functional verification.

References

1. S.T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in *LEET*, vol. 8 (2008), pp. 1–8
2. J. Zhang, Q. Xu, On hardware Trojan design and implementation at register-transfer level, in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2013), pp. 107–112
3. S. Skorobogatov, C. Woods, Breakthrough silicon scanning discovers backdoor in military chip, in *Proc. International Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2012), pp. 23–40
4. Y. Liu, Y. Jin, Y. Makris, Hardware Trojans in wireless cryptographic ICs: silicon demonstration & detection method evaluation, in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 399–404
5. M. Beaumont, B. Hopkins, T. Newby, *Hardware Trojans-Prevention, Detection, Countermeasures (A Literature Review)* (Australian Government Department of Defense, 2011)
6. Defense Science Board Task Force on High Performance Micorchip Supply, Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics (United States Department of Defense, 2005)
7. Y. Jin, N. Kupp, Y. Makris. Experiences in hardware trojan design and implementation, in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust* (2009), pp. 50–57
8. Trust-Hub Website, https://www.trust-hub.org/
9. S. Wei, K. Li, F. Koushanfar, M. Potkonjak, Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry, in *Proceedings of ACM/IEEE Design Automation Conference* (2012), pp. 90–95

10. M. Hicks, M. Finnicum, S.T. King, M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2010), pp. 159–172
11. C. Sturton, M. Hicks, D. Wagner, S. T King, Defeating UCI: building stealthy and malicious hardware, in *Proceedings of the IEEE International Symposium on Security and Privacy (SP)* (2011), pp. 64–77
12. J. Bormann, et al., Complete formal verification of TriCore2 and other processors, in *Design and Verification Conference* (2007)
13. J. Zhang, F. Yuan, L. Wei, Z. Sun, Q. Xu, VeriTrust: verification for hardware trust, in *Proc. IEEE/ACM Design Automation Conference (DAC)* (2013), pp. 1–8
14. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using boolean functional analysis, in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)* (2013), pp. 697–708
15. J. Zhang, F. Yuan, Q. Xu, DeTrust: defeating hardware trust verification with stealthy implicitly-triggered hardware trojans, in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)* (2014), pp. 153–166