Efficient Control-Flow Subgraph Matching for Detecting Hardware Trojans in RTL Models

LUCA PICCOLBONI, Columbia University
ALESSANDRO MENON and GRAZIANO PRAVADELLI, University of Verona

Only few solutions for Hardware Trojan (HT) detection work at Register-Transfer Level (RTL), thus delaying the identification of possible security issues at lower abstraction levels of the design process. In addition, the most of existing approaches work only for specific kinds of HTs. To overcome these limitations, we present a verification approach that detects different types of HTs in RTL models by exploiting an efficient control-flow subgraph matching algorithm. The prototypes of HTs that can be detected are modelled in a library by using Control-Flow Graphs (CFGs) that can be parametrised and extended to cover several variants of Trojan patterns. Experimental results show that our approach is effective and efficient in comparison with other state-of-the-art solutions.

CCS Concepts: • **Security and privacy** → **Embedded systems security**; • **Hardware** → **Functional verification**;

Additional Key Words and Phrases: Hardware trojan detection, RTL models, control-flow sub-graph matching

ACM Reference format:
Luca Piccolboni, Alessandro Menon, and Graziano Pravadelli. 2017. Efficient Control-Flow Subgraph Matching for Detecting Hardware Trojans in RTL Models. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 137 (September 2017), 19 pages.
https://doi.org/10.1145/3126552

1 INTRODUCTION

Globalization of hardware design and fabrication processes have raised serious concerns about hardware-based attacks [13, 21]. Hardware has been always assumed to be the guarantee of trustworthiness in cryptographic algorithms and security protocols. However, several backdoors have been reported in the last years, especially in military contexts [4, 23]. In 2008, for example, a chip installed on a Syrian radar was rumoured to have been tampered to avoid warning in case of air assaults [4]. In 2012, Skorobogatov and Woods [23] discussed the first case of an hardware backdoor found in a chip used in military and industrial cores. Besides that, backdoors have been reported in more familiar contexts as well. In 2012, at the Black Hat Conference [1], it has been showed how to obtain the key of a cryptographic algorithm to lock or unlock hotel rooms. The attacker gained

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.

Authors' addresses: L. Piccolboni was within the Department of Computer Science, University of Verona, Verona, Italy. Now, he is within the Department of Computer Science, Columbia University, New York, NY, USA; email: piccolboni@cs.columbia.edu; A. Menon and G. Pravadelli are within the Department of Computer Science, University of Verona, Verona, Italy; emails: alessandro.menon@studenti.univr.it and graziano.pravadelli@univr.it.

https://doi.org/10.1145/3126552

access to the memory of the hotel cards by resetting their fuse bits. To make it worse, the growing number of today smart devices makes such attacks dangerous and plausible in any context [17].

Among several threat kinds, *Hardware Trojans (HTs)* are malicious alterations of integrated circuits with the aim of disrupting their functional or extra-functional behaviours [5]. They are composed of two main parts: the *trigger* that activates the malicious behaviour under certain conditions, and the *payload* that implements the malicious tasks. The triggers can include (i) functional-based conditions, e.g., a specific value or a sequence of values, which activates the payload once it has been observed on a certain register or port, (ii) physical-based conditions, e.g., reaching a value of temperature or power, or (iii) time-based conditions, e.g., a certain number of cycles or operations that must be counted. Payloads typically exhibit even more diversity, e.g., leakage of information, data corruptions, performance loss, etc.

HTs can be added during every phase of the fabrication process, e.g., design or synthesis, and they are designed to remain silent during the whole verification and testing phase, thus causing the failure of the standard verification approaches. For this reason, several ad-hoc verification techniques have been proposed for the detection of HTs at different levels of abstraction, e.g., [10, 12, 15, 19, 20, 24, 28, 32]. However, especially at the Register-Transfer Level (RTL), the existing solutions are not enough to guarantee the trustworthiness of the Design Under Verification (DUV). In fact, they present different drawbacks. In some cases, they require manual effort from the designer, e.g., [15, 19, 20] or modifications of the designs, e.g., [10]; in other cases, they address only specific types of attacks, e.g., [19, 20]. Indeed, most of them can be applied only at gate level, thus delaying the identification of HTs late in the design process, when removing the HTs would be too expensive. Besides that, HTs are more and more inserted at RTL because, at this level of abstraction, attackers have high flexibility to implement any malicious function [31].

1.1 Contributions

To overcome the previous limitations, we present a verification approach that detects different kinds of HTs in RTL models by exploiting a control-flow subgraph matching algorithm. Our approach does not require either modifications of the DUV code or manual effort. The kinds of HTs that can be detected are represented in a library that can be parametrised and extended to cover different variants of known HTs. In particular, we make the following contributions:

- we define a HT library containing "basic versions" of triggers and payloads of known RTL implementations of HTs; the HT library can be easily extended by users to keep up with new types of triggers and payloads;
- we define a set of directives that allow to automatically and dynamically create more complex variants of the basic versions of the HTs included in our HT library, such that camouflaged instances of those HTs cannot easily escape the detection; this is essential to detect the HTs whose implementations can be recursively extended;
- we implement an approach that exploits a control-flow subgraph matching algorithm to detect the presence of HT variants inside an RTL model; this algorithm flags as suspicious the parts of the code that have the same topological structure (i.e., nodes and edges) of HT variants included in the library and of their variants;
- we define confidence metrics to distinguish the presence of actual HT instances from false positives.

1.2 Organisation

The rest of the paper is organised as follows. Section 2 provides the necessary background. Section 3 presents the HT library. Section 4 details the HT detection approach. Section 5 describes the confidence metrics to discard false positives. Section 6 deals with the experimental results. Section 7 describes the limitations of the approach and discusses possible ways to overcome them. Section 8 summarizes the most-closely related works. Lastly, Section 9 concludes the paper.

2 PRELIMINARIES

We provide the following definitions to formalize the main concepts and the approach presented in the paper.

Definition 1. Given a process P of a DUV at RTL, a Control-Flow Graph (CFG) for P is a tuple $CFG(P) = (B, E, \rho, s, e)$:

- $B = \{b_1, \ldots, b_n\}$ is the finite set of basic blocks, i.e., sequences of consecutive instructions without any branch;
- $E \subseteq B \times B$ is the finite set of edges between the blocks such that $(b_1, b_2) \in E$ if and only if b_2 can be executed after b_1 in at least one of the possible executions of the process P;
- $\rho : E \rightarrow (0, 1]$ is the function such that $\rho(b_1, b_2)$ is the probability that b_2 follows b_1 during an execution of P;
- $s, e \in B$ are the first and last basic blocks respectively.

The CFG for a DUV at RTL is the union of the CFGs of all processes belonging to the DUV.

Definition 2. Given the the following CFGs:

- $CFG(P_1) = (B_1, E_1, \rho_1, s_1, e_1)$ (which is the $pattern$)
- $CFG(P_2) = (B_2, E_2, \rho_2, s_2, e_2)$ (which is the $target$)

then $CFG(P_1)$ is an $\alpha-$subgraph of $CFG(P_2)$ if and only if there exists (i) $h(B_0) \subseteq h(B_2)$ and $E_0 \subseteq E_2$ (which is the $match$) and (ii) a mapping function $\omega : h(B_1) \rightarrow h(B_0)$ such that $\forall (b_1, b_2) \in E_1$ the following conditions apply:

- $(b_1, b_2) \in E_1$ iff $(\omega(b_1), \omega(b_2)) \in E_0$;
- $\mid \rho_1(b_1, b_2) - \rho_2(\omega(b_1), \omega(b_2)) \mid \leq \alpha$.

where $h(B_i)$ is an abstraction function that removes all the instructions from the basic blocks in B_i.

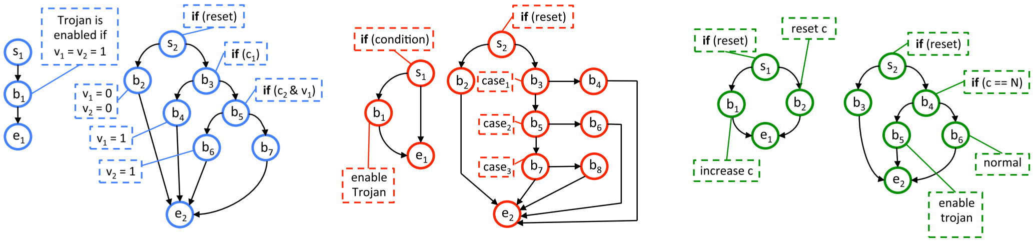
It is worth noting that the abstraction function h allows performing the subgraph matching between two CFGs by only considering their topological structures, while it does not consider the instructions belonging the basic blocks of the CFG. This makes the matching independent from the actual code of the DUV with respect to the HT implementations included in our HT library. The actual instructions in the basic blocks are used only as confidence metrics (Section 5).

2.1 Threat Model

Given the increasing complexity of RTL descriptions, there exist several possibilities to introduce malicious behaviours that can escape traditional RTL verification approaches. In addition, HTs introduced at RTL are generally very hard to be detected. In fact, their detection requires checking the RTL model against the specification. However, the specification is generally incomplete and important details (for example cycle-accurate execution models) are not reported [18].

Similarly to [18–20], three scenarios of rogue insertions at RTL are primarily considered in this paper:

(1) an in-house designer intentionally hides malicious behaviours in the RTL modules before verification and synthesis steps;



(a) cheat code (b) dead machine (c) ticking timebomb

Fig. 1. Examples of the CFGs of HT triggers in the proposed HT Trojan Library. The dashed boxes represent the instructions (in pseudo-code) inserted in the basic blocks associated to the nodes. In case of a node with two outgoing edges, the left and the right edges correspond to the case in which the branching condition evaluated in the node is true and false, respectively.

(2) third-party RTL modules, which have been intentionally manipulated to insert an HT, are integrated in the system under development;
(3) a HT is automatically inserted by a CAD tool. For example, a high-level synthesis tool could add malicious behaviours moving from a TLM to a RTL model; a language translator can do the same when converting from a hierarchical SystemC model to a flattened VHDL description. This risk has been underestimated in the past, by thinking that CAD tool vendors would be ruined if it was discovered that their tools introduce HTs. However, attackers other than the tool vendors can affect the functionality of the tool such that it introduces HTs in the manipulated designs without the vendor's consent, for example, by exploiting their configuration mechanism, which is very often based on scripting.

Given these premises, our paper is intended to present a tool for the automatic detection of HTs introduced at the RTL design stage of the fabrication process. It can be used to address each of the three previous scenarios to detect HTs based on known triggering mechanisms and their variants as reported in the following sections.

3 HARDWARE TROJAN LIBRARY

To make effective the verification approach presented in this paper, we defined a HT library that includes the RTL implementations of known HT triggers and their camouflaged variants. We have currently considered the basic triggers adopted in [22] and [33]. Moreover, we defined a set of directives that permit to automatically create, during the HT detection phase, variants of the basic implementations. This makes our approach more general compared to existing solutions and hard to be defeated by camouflaging the HT basic structures, as it has been done, for example, in [24, 31]. We classified the HT triggers included in the library in three categories: *cheat code*, *dead machines* and *timebombs*.

Trigger 1. A *cheat code* is a value (or sequence of values) that enables the payload when it is observed on a specific location (e.g., a register, which at RTL corresponds to a variable in a particular state) [27]. A cheat code is hard to be detected by using the standard verification approaches because it generally behaves as an extremely rare corner case. Figure 1(a) reports a simple example of CFG of a cheat code's trigger based on two processes. The process on the right implements a sequence of branches that checks if the required cheat code sequence (in this example, the sequence $\langle c_1, c_2 \rangle$) has been read. When the cheat code value c_1 (respectively, c_2) is caught in b_3 (b_5), then the variable v_1 (v_2) is updated to 1 in b_4 (b_6). When both v_1 and v_2 are 1, the process on the left flags the activation to the payload in b_1. The cheat code sequences can be quite long: this complicates the activation of the trigger condition with standard verification approaches, due to the very low probability of exploring the path corresponding to the sequence.

Trigger 2. A *dead machine* activates the HT when specific state-based conditions are met. It can be considered a generalization of cheat codes to create even more complex conditions of activation. Figure 1(b) shows a possible implementation. The process on the left verifies if a global condition is satisfied to activate the payload. The process on the right implements a state machine. Each state is used to evaluate a sub-condition. When all sub-conditions are met the global condition is satisfied. For example, the state machine in Figure 1(b) can be used to count the number of times a set of instructions of a processor has been executed. Basic blocks b_3, b_5, b_7 check three different instructions, while basic blocks b_4, b_6, b_8 update internal variables to count their occurrences in the program executed by the processor. When all these three instructions are executed a specific number of times, the HT's trigger is enabled in b_1. Again, standard verification approaches fail to detect such a trigger due to the huge number of possibilities that must be verified before the trigger becomes active.

Trigger 3. A *ticking timebomb* enables the payload when a certain number N of clock cycles has been counted. Standard verification approaches fail to check this trigger. In fact, if N is sufficiently high, dynamic methods require long simulations, while formal approaches face the state explosion problem. Figure 1(c) illustrates the CFG of a possible implementation of this trigger based on two processes. The process on the left resets the counter whenever the DUV is reset, to avoid being found during functional verification. The process on the right realises a counter and once it reaches the value N the malicious task is enabled (in the block b_5).

The HT library we defined includes a *basic implementation* and a *configuration file* for each of the previous triggers. The basic implementation consists of the simplest form of the trigger's code. The configuration file includes *extension directives* and *confidence directives* to make the triggers parametric. In this way, our detection algorithm (Section 4.2) can identify also more complex variants. These directives are grouped in Figure 2 depending on the purpose.

3.1 Extension Directives

The extension directives are used to automatically modify the CFG of the trigger's basic implementations, during the HT detection process. These directives allow creating more complex variants of the HT's triggers at run time, such that our detection algorithm cannot be easily defeated by changing the structure of a known HT (e.g., by increasing the number of conditions in a cheat code making more difficult its detection, by increasing the number of processes to split the trigger conditions, or by adding irrelevant states in a dead machine, to hide the presence of the trigger). Specifically, these directives are optionally specified by the designers in a configuration file and used by the detection algorithm to extend the CFG of the corresponding trigger implementation at runtime. Each variant created through such directives will be searched in the CFG of the DUV to ensure that neither the basic version nor the variants obtained with such directives are present in the DUV.

For example, suppose that the cheat code depicted in Figure 1(a) is included as a basic implementation in the HT library. The extension directives shown on the right part of Figure 3 can be defined to make it parametric and automatically generate a set of its variants during the detection process. These directives define how many extensions must be recursively done (line 2), and which blocks and edges must be added or removed in the CFG to make the extension at each iteration of the recursion (lines 3–8). The two new nodes, which are added during each extension, are identified in the directives as \$1 and \$2. As a result of this configuration file, at run time, our

Extension Directives	
parametrizable *n*	*n* = 1 if the trigger is parametrizable, *n* = 0 otherwise
bound-number *n*	*n* is the max number of extensions that can be applied
add-basic-blocks *n*	*n* is the number of blocks added at each extension
drop-edge (b_i, b_j)	the edge that goes from block b_i to block b_j is removed
add-edge (b_i, b_j) **p =** *n*	an edge from block b_i to block b_j is added with probability *n* of being traversed
source-basic-block b_i	the block b_i is used as starting point for the extension
up-source-basic-block b_i	b_i is the new starting point of the CFG after the extension

Confidence Directives	
Variable Directives	
var-names $b_1 \dots b_n$	$b_1 \dots b_n$ contain the names of the variables to be checked by the detection algorithm
var-checks $b_1 \dots b_n$	$b_1 \dots b_n$ are checked to **verify** the presence of "relevant" variables *(1)*
up-var-names b_i	add b_i to the blocks containing the names of the variables to be checked
up-var-checks b_i	add b_i to the blocks containing the name of the variables to be reset
Reset Directives	
reset-signal $b_1 \dots b_n$	$b_1 \dots b_n$ are checked to **verify** the presence of the reset signal *(1)*
reset-names $b_1 \dots b_n$	$b_1 \dots b_n$ contain the names of the variables that should be reset
reset-checks $b_1 \dots b_n$	$b_1 \dots b_n$ are checked to **verify** the presence of variables to be reset *(1)*
up-reset-names b_i	add b_i to the blocks containing the names of the variables to be reset
Cheat-code Directives	
cheat-names $b_1 \dots b_n$	$b_1 \dots b_n$ contain the names of the variables used in a cheat code
cheat-checks $b_1 \dots b_n$	$b_1 \dots b_n$ are used to **verify** the presence of variables in a cheat code *(1)*
Timebomb Directives	
timebomb-names b_i	b_i contain the name of the variable used as a counter in a timebomb
timebomb-checks $b_1 \dots b_n$	$b_1 \dots b_n$ are used to **verify** the presence of a counter var in a timebomb *(1)*
Payload Directive	
payload-names $b_1 \dots b_n$	$b_1 \dots b_n$ contain the names of the variables to look for in the payload

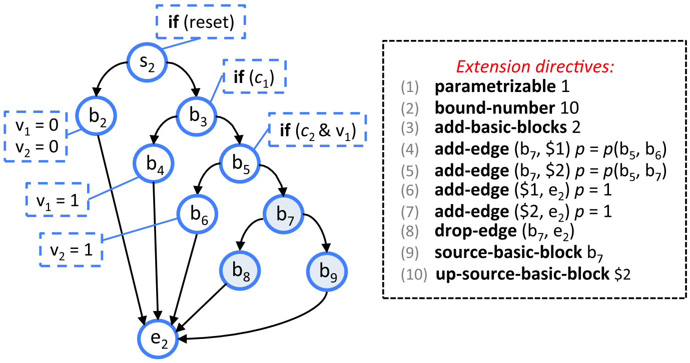
Note: (1) All the basic blocks mentioned in this table are referred to the CFG of the trigger, except for the directives with "**verify**", where blocks refer to the CFG of the DUV. These directives check the presence of certain variables or characteristics in the blocks of the CFG of the DUV that match the basic blocks of the trigger.

Fig. 2. Directives to parametrize the CFGs of the triggers in the HT library.

detection algorithm will search, in the DUV, for the presence of the cheat code, initially, in the form reported in Figure 1(a). Then, it will extend the code as reported on the left part of Figure 3 by adding two new basic blocks (b_8 and b_9 outgoing from b_7) to increase the length of the cheat sequence. Then, two new blocks will be added outgoing from *b*9 (not showed in Figure 3) and so on, till 10 extended versions of the trigger are created (the directive *bound-number* is fixed to 10), and separately checked.

3.2 Confidence Directives

The confidence directives, instead, are used to define the structural characteristics of the trigger's CFG, which must be checked by the detection algorithm to calculate a confidence value, after a possible instance of a HT is found in the DUV. This is used to rank the list of suspicious HT matches



Extension directives:

(1) **parametrizable** 1
(2) **bound-number** 10
(3) **add-basic-blocks** 2
(4) **add-edge** (b_7, \$1) $p = p(b_5, b_6)$
(5) **add-edge** (b_7, \$2) $p = p(b_5, b_7)$
(6) **add-edge** (\$1, e_2) $p = 1$
(7) **add-edge** (\$2, e_2) $p = 1$
(8) **drop-edge** (b_7, e_2)
(9) **source-basic-block** b_7
(10) **up-source-basic-block** \$2

Fig. 3. Use of the extension directives for the parametrization of a cheat code.

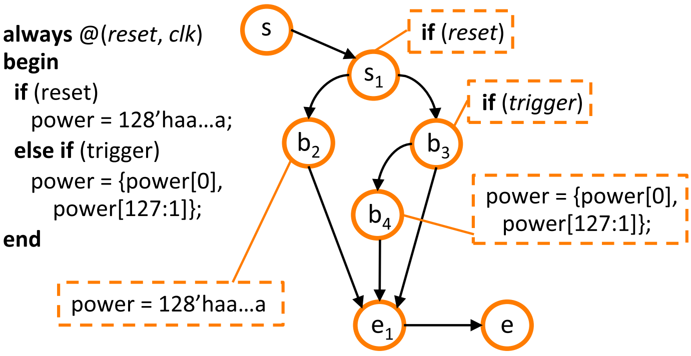

```
always @(reset, clk)
begin
 if (reset)
   power = 128'haa...a;
 else if (trigger)
   power = {power[0],
      power[127:1]};
end
```

Fig. 4. Example of the RTL code and the CFG of an HT payload.

and discard false positives. Further considerations on the confidence directives will be provided in Section 5, after the description of the detection algorithm.

In addition to the list of triggers, the HT library includes the known implementations of *payloads*. They are exploited by the procedure described in Section 5 to calculate the confidence level of the matches identified by the HT detection algorithm. The library includes an implementation for each payload similarly to the case of triggers. An example of payload is reported in Figure 4. It increases the power consumption of the design in which it is inserted by continuously rotating the *power* register (in b_4). The payload is enabled when all the conditions specified in the trigger (not reported in the figure) are met, i.e., when the *trigger* variable becomes *true*.

4 DETECTION OF HARDWARE TROJANS

The overview of our approach is reported in Figure 5. The user provides as input a behavioural RTL model of the DUV (Verilog or VHDL) and the HT library presented in the previous section (optionally extended with different HT triggers and payloads). Our approach returns a *report* that includes potential matches with the HTs in the library. Each match is associated to a confidence value that helps discard the false positives. The approach is based on two main algorithms:

(i) The *extraction algorithm* creates the CFG for the RTL DUV accordingly to Definition 1 (Section 2) for each process included in the DUV. The algorithm creates the CFGs of the triggers and payloads included in the HT library as well. In this way, both the structures

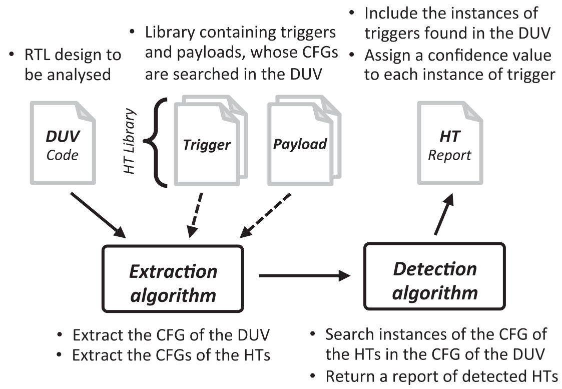

• RTL design to be analysed
• Library containing triggers and payloads, whose CFGs are searched in the DUV
• Include the instances of triggers found in the DUV
• Assign a confidence value to each instance of trigger
DUV Code
HT Library
Trigger
Payload
HT Report
Extraction algorithm
Detection algorithm
• Extract the CFG of the DUV
• Extract the CFGs of the HTs
• Search instances of the CFG of the HTs in the CFG of the DUV
• Return a report of detected HTs

Fig. 5. Overview of the proposed approach.

of the DUV and the HTs are represented with graph-based models that highlight their execution paths and simplify the detection of HTs. This algorithm is presented in Section 4.1;

(ii) The *detection algorithm* identifies the HTs by exploiting a subgraph isomorphism algorithm. First, it searches for the parts of the CFG of the DUV that match the CFGs extracted from the triggers of the HT library. Afterwards, it analyses the matched instances to provide confidence values that help discard the false positives. The confidence values take into account the structural characteristics of the CFGs and the related payloads. This is described in Section 4.2.

4.1 Extraction Algorithm

The extraction algorithm that extracts the CFGs from the DUV and the HT source code starts by obtaining the corresponding syntax trees. A syntax tree represents the design with a tree structure by abstracting the details concerning the syntax of the programming language, but preserving unaltered its semantics. Each node of the tree represents a construct that occurs in the source code. In the proposed approach, the syntax trees of the DUV and HTs are created with HIFSuite [6]. This guarantees to be independent from a particular programming language and makes our approach more general. Additionally, the use of HIFSuite ensures that hierarchical structural designs are flattened, and this considerably simplifies the detection of HTs in hierarchical designs. The syntax tree is then recursively analysed and the CFGs of the RTL processes are created in according with Definition 1. Specifically, the extraction algorithm analyses sequentially the syntax tree for each process of the target code. It creates the first and the last basic block and it maintains a reference to the current basic block. After that, it reads sequentially all the instructions of the process. The instructions that do not create branches are added to the current block, while for the others the following rules apply:

- *if-then(-else)*: the branch condition is inserted into the current basic block; a block is created for the *then* path and, optionally, another one is created for the *else* path. The visit proceeds recursively on the two paths, by updating the current block and by creating the necessary edges as depicted in Figure 6(a) and (b)[1]. The edges are labelled with the probability of taking the corresponding branches. In Figure 6, p_t, p_e, p_{nt} represent, respectively, the probability

[1] The left edge of a basic block with a branch condition refers to the then path, while the right edge is the else path.

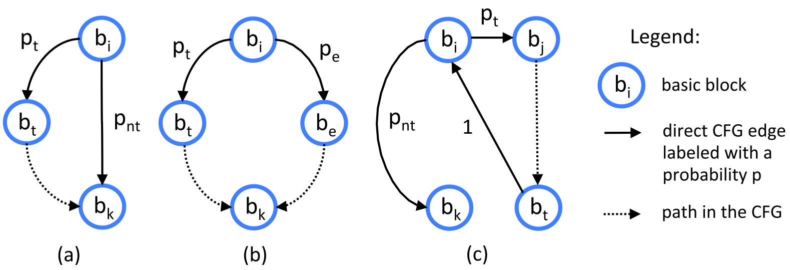


Fig. 6. CFGs for (a) an *if-then*, (b) an *if-then-else* and (c) a *for* statements.

of traversing the *then*, *else* and *not-then* (for cases without *else*) paths. Note that with this rule we cover other constructs that create branching conditions, such as *switch* and *ternary operators*. We handle these cases in the same way, to avoid camouflaging that change the syntax, but not the semantics of the corresponding code;

- *for*: a new block, b_i, is created for the branch condition as illustrated in Figure 6(c); the visit proceeds recursively on the internal instructions of the loop with b_j and b_t that represent the first and the last basic block of the visit; the corresponding edges are created and finally the current block is updated to b_k, i.e., the first block after the loop. Again, the edges are labelled with the probability of reaching the destination node, and this rule covers also other possible variants of expressing loops in the code, which are not semantically different.

The probability associated to the edges is one of the four characteristics evaluated to measure the confidence level when a possible HT match is detected in the DUV, as described in Section 5. To *statically* determine the probability of taking a branch, static analysis and slicing approaches should be used. In fact, it is necessary to determine the dependencies among all the variables involved in the conditions of each branch. However, current slicing algorithms do not scale well with design complexity. Therefore, an approach based on a SMT constraint solver is adopted. Differently from a SAT solver, which works only on Boolean values, an SMT solver can solve constraints according to several theories like integers, bit vectors, etc. [8]. The solver is used to calculate the total number of models, i.e., assignments to the variables that satisfy the conditions of the branches. By dividing the number of models by the number of all the possible assignments, it is possible to determine the probabilities. This approach is more scalable with respect to program slicing, even if it uses an exponential algorithm, because the expressions that can be found in the condition of branches of the DUV are often simple and composed of up to three or four variables. Determining the dependencies among the variables in program slicing is much more expensive because all the instructions of the DUV could be evaluated. Indeed, to further increase the scalability, the solutions of simple conditions, e.g., $x = 10$, $z < w$, etc., are determined by using the ranges of values of the variables involved in such conditions, instead of calling the SMT solver.

4.2 Detection Algorithm

The algorithm that detects the presence of HTs in the DUV performs two steps. First, it tries to match instances of the abstract CFGs of the HT triggers in the abstract CFG of the DUV. The matching is performed after the h abstraction function of Definition 2 has been applied to generate the abstract versions of the CFGs (all the instructions from the basic blocks are removed). Second, the algorithm calculates a confidence value representing how much it is likely that each matched instance is a HT. The first step is described hereafter, while the second step is discussed in Section 5.

```
1 procedure MATCH-TRIGGERS (duv, HTLibrary)
2     targets = EXTRACT-CFG(duv);
3     foreach trigger in HTLibrary do
4         pattern = EXTRACT-CFG(trigger); count = 0;
5         while count < pattern.getMaxBound() do
6             matches ∪= SEARCH(pattern, targets);
7             pattern.augmentSize(); count++;
8     CALC-CONF(duv, matches, HTLibrary);
```

Fig. 7. Detect HTs with subgraph matching.

```
procedure CALC-CONF (duv, matches, HTLibrary)
    foreach payload in HTLibrary do
        payloads ∪= EXTRACT-CFG(payload);
    foreach match in matches do
        match.conf = $\alpha_1$ * CHECK-VARIABLES(match);
        match.conf += $\alpha_2$ * CHECK-RESETLOGIC(match);
        match.conf += $\alpha_3$ * CHECK-PROBABILITIES(match);
        match.conf += $\alpha_4$ * CHECK-PAYLOADS(match, duv);
```

Fig. 8. Calculate the confidence of matches.

The detection algorithm is showed in Figure 7. The algorithm starts by extracting the CFG of the DUV with the procedure presented in Section 4.1 (line 2). In the same way, for all the triggers defined in the HT library, it extracts the corresponding CFGs (lines 3–4). After that, the algorithm calls the RI subgraph isomorphism algorithm described in [7] (line 6) to find all the subgraphs in the abstract CFGs of the DUV that match the abstract CFG of the current trigger, accordingly to Definition 2. Note that during this phase, the instructions in the basic blocks of the CFGs are not considered for the matching (see function *h* in Definition 2). In fact, this algorithm identifies only the parts in the DUV that are topologically similar, i.e., they have the same structure of edges and nodes, of the HTs. The instructions are only considered later during the calculation of the confidence value. If the trigger's CFG is parametrizable, it is properly extended by modifying its CFG (line 7) as required by the extension directives included in the configuration file of the HT library. In this way, the matching algorithm is repeated for different trigger variants up to a fixed number of times (line 5). Finally, all the returned matches are evaluated with the procedure described in Section 5 to determine a confidence measure used to discard false positives.

4.2.1 *Guarantees.*

The matching algorithm guarantees that:

(1) the basic version of a given HT trigger is identified ($count = 0$);
(2) the extensions of the HT trigger obtained with the extension directives (Section 3) are identified ($count > 0$);
(3) any attempt of obfuscating the HT by creating a hierarchical structural design is ineffective, since the CFG of the DUV is extracted from the flattened version of the RTL model obtained by using HIFSuite.

Clearly, we fix a maximum bound for the application of the extension directives (line 5) to allow the algorithm to terminate. Thus, this implies that only the extended versions of the HT trigger obtained with a number of extensions lower than the bound are identified.

4.2.2 *Complexity.*

The complexity of the matching algorithm is:

$$O(l_{HT} * b_{HT} * C(n_{DUV}, n_{HT}))$$

where l_{HT} is the total number of triggers in the HT library, b_{HT} is the maximum number of extensions of the HT triggers, n_{DUV} is the number of nodes in the DUV, n_{HT} is the maximum number of nodes of the HT triggers, and the function C represents the complexity of the subgraph isomorphism algorithm, which is exponential in the general case. In fact, the subgraph isomorphism problem is NP-complete, but the RI algorithm has been showed to be especially efficient with *sparse graphs*[2], and, indeed, CFGs are often sparse. Besides that, the size of the CFGs of HT triggers, i.e.,

[2] A graph is sparse if the ratio between the number of edges and the number of vertexes is high.

n_{HT}, is typically small, since they are designed to be hidden in the code of the DUV. Thus, searching an instance of a small graph (the HT trigger) inside a sparse graph (the DUV) guarantees the overall scalability of our approach, as shown in Section 6.

5 MATCHING CONFIDENCE

Figure 8 depicts the algorithm that returns the confidence values of the triggers found by the procedure described in Section 4.2. For the sake of clarity, in the following, the term *match* refers to the trigger instances returned by the procedure of Figure 7, while *pattern* refers to the trigger instances of the HT library. The algorithm in Figure 8 takes in inputs the DUV, the matches, and the HT library. It returns for each match a confidence value in the range [0, 1], where 1 indicates the highest confidence level. The algorithm starts by extracting the CFGs of the payloads of the HT library (lines 2–3). After that, it evaluates the following characteristics to determine the confidence value of the match:

c_1: *presence of variables* (line 5): the algorithm verifies if the match uses some variables *in the same way* of the corresponding pattern. For example, in the case of Trigger 3 it checks if there is a variable used as a counter in the match, but not necessarily with the same name of one in the trigger. The CHECK-VARIABLES function returns the ratio between the number of variables found in the same basic blocks of the pattern and the number of variables specified with the *var. directives* of Figure 2. In particular, the directive **var-names** defines where are the names of the variables that must be considered (to not depend on the names used in the pattern) and the directive **var-checks** specifies the basic blocks to be checked. The other two variable directives in Figure 2 are used to update these blocks in case of extensions. For specific triggers other directives are used to check how the variables are used, e.g., where the counter is increased (*time directives*) or the cheat code variables are updated (*cheat directives*).

c_2: *presence of the reset logic* (line 6): the algorithm checks if the match has a reset logic similar to the reset logic of the pattern. For example, in the case of Trigger 3, it checks if a variable (counter) is reset whenever the DUV is reset. The CHECK-RESETLOGIC function returns a value in the range [0, 1] depending on the number of characteristics that are identified: i.e., presence of the reset signal and variables that are reset and used in the same blocks. The *reset directives* in Figure 2 are used to specify which are the basic blocks and the variables to be considered.

c_3: *average distance of the probabilities of the match and the corresponding pattern* (line 7): the algorithm calculates the distance between the probabilities of traversing each edge in the match and the expected probabilities for traversing the corresponding edges in the pattern; nearer are the probabilities, more is likely that the match is a real instance of the pattern. Several triggers create branches with low or high probabilities, making this characteristic useful to evaluate their confidence (for example Trigger 1). The CHECK-PROBABILITIES function returns a value in the range [0, 1], where 1 is used to indicate a perfect correspondence of the probabilities.

c_4: *degree of dependence between the match and the most affine payload* (line 8): the algorithm verifies if there are shared variables, i.e., registers, between the match and one of the payload specified in the HT library. Specifically, the CHECK-PAYLOADS calculates the ratio between the number of shared variables that have been identified and the number of variables defined with the *payload directives* in the configuration file. This permits to augment the confidence of the matches that have a payload in the DUV. Each trigger is checked against each payload instance to find the most affine, i.e., the payload that has the highest ratio for that trigger.

Table 1. Main characteristics of the Trush-HUB benchmarks

Name	*Trigger*	*Payload*	*Blocks*	*Edges*	*Name*	*Trigger*	*Payload*	*Blocks*	*Edges*
AES-T400	Cheat code	info leakage	2118	3176	RS232-T100	Cheat code	stuck-at 0/1	134	200
AES-T500	Cheat code	power waste	2109	3165	RS232-T200	Time bomb	reliability issue	130	193
AES-T600	Cheat code	info leakage	2114	3170	RS232-T300	Cheat code	info leakage	132	195
AES-T700	Cheat code	info leakage	2112	3166	RS232-T400	Time bomb	info leakage	130	192
AES-T800	Cheat code	info leakage	2117	3174	RS232-T500	Cheat code	stuck-at 0/1	132	195
AES-T900	Time bomb	info leakage	2114	3168	RS232-T600	Dead machine	stuck-at 0/1	157	233
AES-T1000	Cheat code	info leakage	2109	3162	RS232-T700	Dead machine	stuck-at 0/1	155	230
AES-T1100	Cheat code	info leakage	2114	3170	RS232-T800	Cheat code	reliability issue	124	184
AES-T1200	Time bomb	info leakage	2113	3168	RS232-T900	Dead machine	reliability issue	159	236
AES-T1300	Cheat code	info leakage	2141	3221	RS232-T901	Cheat code	reliability issue	159	236
AES-T1400	Cheat code	info leakage	2150	3236	BasicRSA-T100	Cheat code	info leakage	81	119
AES-T1500	Time bomb	info leakage	2148	3232	BasicRSA-T200	Cheat code	reliability issue	81	120
AES-T1600	Cheat code	info leakage	2127	3191	BasicRSA-T300	Time bomb	info leakage	92	137
AES-T1700	Time bomb	info leakage	2114	3169	BasicRSA-T400	Time bomb	info leakage	93	139
AES-T1800	Cheat code	power waste	2101	3152					
AES-T1900	Time bomb	power waste	2106	3160					

Table 2. Main characteristics of the *Cryptoplatform*

Name	*Trigger*	*Payload*	*Blocks*	*Edges*
Crypto-T000	N/A	N/A	4402	6503
Crypto-T100	Time bomb	info leakage	4424	6537
Crypto-T100	Cheat code	info leakage	4424	6537
Crypto-T200	Time bomb	info leakage	4416	6525
Crypto-T300	Time bomb	info leakage	4410	6519
Crypto-T400	Cheat code	info leakage	4406	6516

Finally, these four characteristics are combined with a linear combination to calculate the confidence value β of each match, i.e. $\beta = \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3 + \alpha_4 c_4$ where the constants α_i depend on the trigger, such that $\sum_i \alpha_i = 1$.

6 EXPERIMENTAL RESULTS

The effectiveness of our approach has been evaluated on a set of RTL benchmarks from Trust-HUB [3] and on the RTL implementation of the *Cryptoplatform* that includes some cryptographic cores (*aes*, *camellia*, *des*, *sha*, *xtea*), a *plasma-cpu* and a *memory* from OpenCores [2]. Tables 1 and 2 shows the characteristics of these benchmarks. They report the name, the types of triggers and payloads and the number of basic blocks and edges of their CFGs. Each benchmark includes one HT, except for Crypto-T000 that is a Trojan-free instance of the Cryptoplatform, and Crypto-T100 that hides two HTs.

Each benchmark is verified against the HT library we defined in Section 3. The characteristics of the library are reported in Table 3. The tables show the CFG size of the different instances in terms of number of basic blocks and edges. We included different implementations of the same triggers because they cannot be obtained from the others by applying only the defined directives. In fact, even if the triggers are similar, their implementations are topologically different from the others. The table reporting the payloads includes also a brief description of their effects when inserted in the DUV.

Table 3. Main characteristics of the triggers and payloads in the HT Library

Triggers		
Name	*Blocks*	*Edges*
cheat-T001	4	4
cheat-T002	5	6
cheat-T003	6	7
cheat-T004	16	21
cheat-T005	11	14
cheat-T006	11	14
mach-T001	10	11
mach-T002	11	13
timeb-T001	13	16
timeb-T002	14	19
timeb-T003	12	15
timeb-T004	6	7
timeb-T005	14	17

Payloads			
Name	*Brief Description*	*Blocks*	*Edges*
payload-T001	transmits critical information	16	21
payload-T002	increases power consumption	8	9
payload-T003	steals information (covert channel)	10	13
payload-T004	steals information (leakage current)	12	15
payload-T005	changes the memory addresses	7	7
payload-T006	manipulates the output signals	7	7

6.1 Experimental Evaluation

To verify the effectiveness of our approach we used the following evaluation strategy. The HT library reported in Table 3 contains the same *categories* (but not the same code) of HTs included in the benchmarks described in Table 1. The HT library and the HTs injected in the considered benchmarks derive both from Trust-HUB. This is necessary because our approach can detect only the known categories of HTs that are inserted in the HT library, including their variants obtained by applying the rules defined in Section 3 (the implications of this observation are discussed in details in Section 7). Thus, we decoupled the triggers of Trust-HUB from the corresponding payloads, and we included the basic versions of the triggers in our library. Note also that our algorithms do not depend on the actual implementations of the specific HTs. The detection algorithm considers only the topological structure of the CFGs of the HTs without taking into account the instructions in the CFGs (Section 4.2). Indeed, the algorithm that assigns the confidence does not depend on the names and the specific instructions used in HTs. In fact, it tries to discover behaviours (Section 5).

With such a premise, we expect that our verification approach detects all the HTs included in the considered benchmarks, as they are known by the HT library. Thus, our goal is to show that our verification approach can help users to distinguish actual HTs from the false positives and classify them accordingly to their respective categories.

6.2 Trust-HUB Benchmarks

The proposed approach has been compared with three state-of-the-art methodologies presented in [10, 19, 20]. The results of these experiments are reported in Table 4. The first column reports the names of the benchmarks. The following three columns report if the HTs can be detected by the three considered state-of-the-art techniques. The rest of the columns report the results for the proposed approach. Specifically, they show if the approach is able or not to detect the HTs (*Found*), the number of identified matches (*Matches*), the confidence level of the match corresponding to the HT actually present in the design (β_{HT}), the highest confidence level among the matches corresponding to false positives (β_{max}), the number of matches with a confidence that is higher than the confidence of the HT actually present in the design (*FP*), and lastly the time in seconds required by the proposed approach ($T(s)$). The techniques proposed in [20] and [19] are specialized for specific kinds of payloads: register corruptions and information leakage respectively. Thus, they are able to find only a limited number of HTs and they do not guarantee the trustworthiness

Table 4. Results: Trust-HUB Benchmarks

	State of the art			**Proposed Approach**					
Name	[20]	[19]	[10]	*Found*	*Matches*	β_{HT}	β_{max}	*FP*	*T(s)*
AES-T400	no	no	~	yes	3	0.95	0.64	0	5.04
AES-T500	no	no	~	yes	7	0.93	0.68	0	4.80
AES-T600	no	yes	~	yes	5	0.93	0.41	0	5.12
AES-T700	yes	yes	~	yes	5	0.85	0.50	0	5.11
AES-T800	yes	yes	~	yes	9	0.93	0.65	0	5.04
AES-T900	no	yes	~	yes	7	0.95	0.62	0	4.78
AES-T1000	no	yes	~	yes	4	1.00	0.64	0	4.76
AES-T1100	yes	yes	~	yes	5	0.94	0.47	0	5.67
AES-T1200	no	yes	~	yes	4	0.96	0.54	0	4.69
AES-T1300	no	no	~	yes	82	1.00	0.65	0	5.62
AES-T1400	no	no	~	yes	81	0.99	0.69	0	4.85
AES-T1500	no	no	~	yes	83	0.98	0.65	0	5.80
AES-T1600	no	no	~	yes	7	0.96	0.54	0	4.86
AES-T1700	no	no	~	yes	3	0.98	0.63	0	5.38
AES-T1800	no	no	~	yes	9	1.00	0.69	0	4.86
AES-T1900	no	no	~	yes	11	0.97	0.72	0	4.82
AES-T2000	no	yes	~	yes	6	0.93	0.41	0	4.56
AES-T2100	no	yes	~	yes	5	0.95	0.75	0	4.75
RS232-T100	no	no	yes	yes	7	0.36	0.50	2	4.12
RS232-T200	no	no	~	yes	8	0.92	0.56	0	3.13
RS232-T300	no	no	yes	yes	6	0.92	0.31	0	2.74
RS232-T400	no	no	yes	yes	8	0.56	0.51	0	2.32
RS232-T500	no	no	yes	yes	6	0.93	0.31	0	2.80
RS232-T600	no	no	yes	yes	11	0.67	0.35	0	2.39
RS232-T700	no	no	yes	yes	11	0.67	0.53	0	2.58
RS232-T800	no	no	yes	yes	7	0.36	0.50	2	3.23
RS232-T900	no	no	yes	yes	11	0.67	0.52	0	2.43
RS232-T901	no	no	yes	yes	11	0.67	0.52	0	2.48
BasicRSA-T100	no	yes	yes	yes	4	0.25	0.25	3	1.13
BasicRSA-T200	no	no	yes	yes	3	0.25	0.25	1	1.45
BasicRSA-T300	no	yes	yes	yes	4	1.00	0.42	0	1.41
BasicRSA-T400	no	no	yes	yes	5	0.96	0.52	0	1.46

~ depends if activated in the learning phase [10].

Table 5. Results: *Cryptoplatform*

	Proposed Approach					
Name	*Found*	*Matches*	β_{HT}	β_{max}	*FP*	*T(s)*
Crypto-T000	no	23	N/A	0.35	N/A	11.80
Crypto-T100	yes	34	0.81	0.39	0	12.88
Crypto-T100	yes	34	0.72	0.39	0	12.88
Crypto-T200	yes	31	0.96	0.71	0	13.43
Crypto-T300	yes	42	0.88	0.29	0	15.03
Crypto-T400	yes	34	0.90	0.50	0	15.67

of the designs. Furthermore, they require (i) manual efforts for defining the properties that represent the corresponding payloads and (ii) the use of model checking that faces some limitations with temporal-based triggers. On the other hand, the approach proposed in [10] covers a wider range of HTs. Unfortunately, the approach in [10] (i) does not guarantee to find all the HTs, since it adopts some heuristics during the identification of triggers, and (ii) requires modifications to the DUVs (with up to 15.25% of area overhead for the additional circuits added to the designs). Nevertheless, as expected, with the proposed approach we identified all the HTs. Indeed, our approach takes only few seconds, without requiring manual efforts or design modifications. The number of false positives is limited in almost all the cases except for few cases where the basic version of the cheat code (composed of a single value) is detected several times (AES-T1300, AES-T1400, AES-T1500). However, in these few cases, by looking at the confidence levels reported in Table 4, we can see that the actual HT has a confidence level (β_{HT}) whose value is at least 30% higher than the highest confidence level (β_{max}) of the false positives. In general, by looking at the whole Table 4, the confidence values of the matches corresponding to actual HTs are between 21% and 67% higher with respect to the false-positive matches, except for few cases (BasicRSA-T100/T200, RS232-T100/T800). In these special cases, the basic version of the cheat code reports lower confidence values due to the low numbers of characteristics that can be evaluated on their CFGs (composed by only one branch). But, note that even in these cases, the number of matches that have a confidence higher than the confidence of the HT is limited to three. The confidence value has been calculated by using the formula reported in Section 5. The different triggers use different values for the constants α_i. Specifically, in the case of cheat codes, the constant with higher values correspond to the condition c_4 and c_3. For the ticking timebombs, we gave more importance to the presence of the counter in particular nodes (i.e., *time-directives*) and finally, for the dead machine we used a uniform distribution of the different conditions.

6.3 Cryptoplatform

Table 5 reports the results for the Cryptoplatform[3]. The format of the table for the results regarding our approach is the same used in Table 4. The first benchmark confirms that the algorithm that assigns the confidence to each match is effective because all the matches receive a very low value (0.35 or lower) because there are no HTs. The other benchmarks show that the proposed approach returns a limited number of false positive, and their confidence level is low compared to β_{HT}, regardless of the number of the designs in the Cryptoplatform to be analysed. The last column of

[3] [10, 19, 20] were not available for a comparison on the Cryptoplatform.

Table 5 shows that our approach scales well in case of more complex designs than the Trust-HUB benchmarks.

As a final consideration it is worth noting that the HT included in *Cripto-T300* and *Cripto-T400* are particular difficult to be detected, as they do not present a suspicious aspect that generally characterizes HT triggers, i.e. having branches with unbalanced probabilities of being traversed. The low-probable branches are generally associated to the HT activation conditions, thus making their identification easier. To remove this characteristic, the HT of *Cripto-T300* includes a variant of the time bomb trigger that uses more than a counter to activate the trigger condition. This approach was used in [31] to defeat UCI. The HT included in *Cripto-T400*, instead, derives the trigger cheat code from multiple clock cycles, thus making again similar the branches probabilities. However, our approach can detect both of them.

7 LIMITATIONS AND EXTENSIONS

In our experimental results, we first consider the Trust-HUB benchmarks to make a fair comparison with the state-of-the-art methodologies proposed in [10, 19, 20] (Section 6.2). In addition, we showed that our approach works well also with different benchmarks, e.g., the Cryptoplatform (Section 6.3). It is worth to note, however, that insertions of HTs can be unpredictable and unexpected [30]. Our approach exhibits two main limitations in this regard. First, if a HT is not included in the library we defined, our approach can detect neither it nor its variants. This is also the case for software programs where antivirus can reveal only known malwares, indeed. However, note that while software malwares are often distributed, the distribution of HTs is a less common practice today. The research community is working on developing and classifying new threats, e.g., [5, 24, 25, 29, 31, 33], but a lot of work is to be done. Second, if a HT results to be very similar to actual legal code, our approach can result in false positives. This is especially true in the case of HTs based on very simple trigger conditions, e.g., a cheat code looks like an if-else construct (or a sequence of if-else constructs). We showed that our approach already works well since the number of false positives is very limited in most cases. However, for these types of HTs our approach is not currently able to clearly discriminate an HT from the actual legal code (see RS232-T100/T800, BasicRSA-T100/T200 in Table 4). We believe that by augmenting the number of characteristics taken in consideration to calculate the confidence of a match (Section 5) it is possible to significantly reduce the false positives. Indeed, we set the weights of each confidence characteristic experimentally. In future work, we plan to use a machine learning algorithm to find the best weights of the characteristics to discriminate the HTs from the false positives.

Despite such limitations, one main contribution of our work is that the approach we propose can be adapted for the detection of new threats, without being dependent on a specific HT trigger or payload. Our HT library is dynamic; it currently contains implementations of known HTs, but it can be easily extended by the user, in case of new threats, without the need of recoding the tool. On the contrary, many state-of-the-art approaches focus only on specific HTs and they are not easily extendible. To extend the library for a new HT, the user has to provide the basic model of the HT in Verilog or VHDL at RTL. In addition he/she can specify transformation rules (see Figure 2) to let the tool automatically covers different variants of the new HT. This guarantees that the users can customize the HT library depending on their own needs.

8 RELATED WORK

Several methodologies have been proposed for detecting HTs. Trigger conditions limit the possibility of using the standard verification techniques. For example, functional-based triggers require that dynamic techniques generate an exponential number of values to find those that activate the HTs, and time-based triggers can lead to the state explosion problem with formal techniques.

Thus, several specific methodologies have been proposed to detect HTs in both pre-silicon and post-silicon design stages [21]. Since inserting HTs before fabrication is more likely [32], this paper addresses pre-silicon detection: the approaches are classified in *structural-analysis* and *formal techniques* [20].

Structural-analysis techniques perform (coverage) analysis to identify the areas of the DUV netlist that potentially hide HTs. For instance, *FANCI* [28] flags as suspicious the signals of a gate-level netlist that weakly affect outputs because triggers of HTs typically have a weak impact on output ports. *FANCI* calculates a control value that represents the impact between an input and an output that depends on it, by means of truth tables. A cut-off threshold is selected to determine the HTs. *VeriTrust* [32] identifies, on a gate-level netlist, circuits that potentially trigger HTs by determining the circuits that remain dormant during functional verification. *VeriTrust* can produce false positives, i.e., false alarms, in case of incomplete functional verification. It has been showed that both *FANCI* and *VeriTrust* can be defeated by manipulating the trigger design [33]. On the other hand, our approach can be easily adapted to such manipulations, thanks to the directives we introduced in Section 3. Haider et al. [10] proposed an alternative approach, called *HaTCh*, that starts from a gate-level netlist and inserts additional circuits to prevent the HT manifestation. It detects untrusted entities through functional testing and adds specific circuits to warn in case of activation. This approach requires that some HTs must not be activated during the learning phase [10]. On the other hand, our approach does not have this limitation, and it is able to detect the same types of HTs (as highlighted in Table 4). A high-level algorithm that can be applied at structural RTL is *UCI* [12]. It identifies, as suspicious, the portions of a design that go unused during the testing phase and flags at runtime their activation. However, *UCI* has already been demonstrated to be defeated in [24].

To guarantee the design trustworthiness formal techniques have been proposed as well [9]. These methods apply well-known verification strategies, e.g., theorem proving, model checking or equivalence checking, to find the HTs. For example, Love et al. [15] defined a protocol for the acquisition of Intellectual Properties (IPs). The idea is that consumers and vendors agree on a set of "security properties", and the IPs are delivered with a formal proof of them. In this way, consumers can verify the trustworthiness of their IPs with a theorem prover. This technique requires the manual definition of properties. This can be a time-consuming and error-prone task. Our approach is somewhat similar since it can be effectively used to verify third-party IPs. However, it does require minimal manual efforts in defining the directives that characterize the HTs (the extraction and matching of CFGs are automatic), and no definition of properties is required. Rajendran et al. proposed to use a model checker to find (i) malicious corruptions of critical registers [20] and (ii) leakages of critical information [19]. In these cases, formal properties that represent such situations are defined on a formal model of the design and verified by using bounded model checkers. With respect to these works, our approach is intended to be more general (as illustrated in Table 4). Thanks to the definition of the HT library, our approach can be extended to verify the presence of different kinds of threats. Additionally, by looking at the CFG of the design, our approach can detect HTs that have time-based triggers that are complex to handle for model checkers.

In addition to the methodologies described above, other methods have been proposed for *preventing* [27] or detecting at *run-time* [11, 14, 16, 26] HTs. For example, Waksman et al. [27] proposed a set of techniques to silence two common HT triggers: cheat codes and ticking timebombs. The idea is to insert additional logics to prevent the activation conditions of the HTs. Since an exhaustive detection of HTs before synthesis is too hard, many researchers proposed to use runtime approaches to detect HTs after deployment. The idea is to insert additional hardware or software to detect the activation of HTs during the execution. For example, Kim et al. [14] proposed to insert a Dynamic Function Verification (DFV) controller that detects the idle state of untrusted IPs in a System-on-Chip (SoC) and verifies them during the idle state.

9 CONCLUDING REMARKS

This paper presented an automatic verification approach to detect HTs on RTL designs. It exploits the structural characteristics of the CFGs of HT triggers and payloads to localize them in the source code of the DUV. To make our approach as much generic as possible, we defined a HT library that includes basic implementations of different categories of known HTs. This library guarantees that our approach is independent from the types of HTs, as opposed to other state-of-the-art techniques that are able to detect only a specific subset of threats. In fact, while our approach can be easily extended to keep up with new threats (it is sufficient to add the RTL models of the threats in the library), most of the other state-of-the-art techniques must be extensively modified. Additionally, the HT library includes a set of characterization directives, which allow the detection algorithm to dynamically create variants of the basic implementations of HTs at run time. In this way, our approach cannot be easily defeated by camouflaging or obfuscation.

We evaluated the efficiency and effectiveness of our approach by first considering the Trust-HUB benchmarks. Our approach is able to detect all the HTs, as opposed to other techniques in literature, thanks to the flexibility provided by the HT library. Additionally, we showed that our approach can be applied to different benchmarks. The Cryptoplatform is a more complex benchmark that contains different variations of HTs with respect to those included in the Trust-HUB benchmarks. Lastly, our approach is able to detect the HTs in few seconds, making it an efficient alternative for the detection of HTs, as opposed to other techniques which do not scale with the complexity of the benchmarks.

REFERENCES

[1] *My Arduino can beat up your hotel room lock*. http://demoseen.com/bhtalk2.pdf.

[2] *OpenCore*. http://opencores.org/.

[3] *Trust-HUB*. https://www.trust-hub.org/.

[4] S. Adee. 2008. The Hunt for the Kill Switch. In *IEEE Spectrum*.

[5] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. 2014. Hardware Trojan Attacks: Threat Analysis and Countermeasures. In *Proc. of the IEEE*.

[6] N. Bombieri, G. Di Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli. 2010. HIFSuite: tools for HDL code conversion and manipulation. *EURASIP Journal on Embedded Systems* (2010).

[7] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro. 2016. On the Variable Ordering in Subgraph Isomorphism Algorithms. *ACM/IEEE Transactions on Computational Biology and Bioinformatics* (2016).

[8] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proc. of the ACM International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[9] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra. 2015. Pre-silicon security verification and validation: A formal perspective. In *Proc. of the ACM/IEEE Annual Design Automation Conference (DAC)*.

[10] S. K. Haider, C. Jin, M. Ahmad, D. M. Shila, O. Khan, and M. Van Dijk. 2014. HaTCh: Hardware Trojan Catcher. In *Cryptology ePrint Archive*.

[11] S. R. Hasan, C. A. Kamhoua, K. A. Kwiat, and L. Njilla. 2016. Translating circuit behavior manifestations of hardware Trojans using model checkers into run-time Trojan detection monitors. In *Proc. of the IEEE Asian Hardware-Oriented Security and Trust (AsianHOST)*.

[12] M. Hicks, M. Finnicum, S. T. King, M. K. M. Milo, and J. M. Smith 2010. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Proc. of the ACM/IEEE Symposium on Security and Privacy (SP)*.

[13] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor. 2010. Trustworthy Hardware: Identifying and Classifying Hardware Trojans. *Computer* (2010).

[14] L. W. Kim and J. D. Villasenor. 2015. Dynamic Function Verification for System on Chip Security Against Hardware-Based Attacks. *IEEE Transactions on Reliability* (2015).

[15] E. Love, Y. Jin, and Y. Makris. 2012. Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition. (2012).

[16] D. McIntyre, F. Wolff, C. Papachristou, and S. Bhunia. 2010. Trustworthy Computing in a Multi-core System using Distributed Scheduling. In *Proc. of the IEEE International On-Line Testing Symposium (IOLTS)*.

[17] S. Mitra, H.-S. P. Wong, and S. Wong. 2015. Stopping Hardware Trojans in Their Tracks. In *IEEE Spectrum*.

[18] I. Polian, G. T. Becker, and F. Regazzoni. 2016. Trojans in Early Design Steps – An Emerging Threat. In *Proc. of the Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE)*.

[19] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri. 2016. Formal Security Verification of Third Party Intellectual Property Cores for Information Leakage. In *Proc. of the ACM/IEEE International Conference on VLSI Design (VLSID)*.

[20] J. Rajendran, V. Vedula, and R. Karri. 2015. Detecting Malicious Modifications of Data in Third-party Intellectual Property Cores. In *Proc. of the ACM/IEEE Annual Design Automation Conference (DAC)*.

[21] M. Rostami, F. Koushanfar, and R. Karri. 2014. A Primer on Hardware Security: Models, Methods, and Metrics. In *Proc. of the IEEE*.

[22] H. Salmani, M. Tehranipoor, and R. Karri. 2013. On Design Vulnerability Analysis and Trust Benchmarks Development. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*.

[23] S. Skorobogatov and C. Woods. 2012. Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In *Proc. of the ACM International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

[24] C. Sturton, M. Hicks, D. Wagner, and S. K. King. 2011. Defeating UCI: Building Stealthy and Malicious Hardware. In *Proc. of the ACM/IEEE Symposium on Security and Privacy (SP)*.

[25] M. Tehranipoor and F. Koushanfar. 2010. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Design Test of Computers* (2010).

[26] A. Waksman and S. Sethumadhavan. 2010. Tamper Evident Microprocessors. In *Proc. of the ACM/IEEE Symposium on Security and Privacy (SP)*.

[27] A. Waksman and S. Sethumadhavan. 2011. Silencing Hardware Backdoors. In *Proc. of the ACM/IEEE Symposium on Security and Privacy (SP)*.

[28] A. Waksman, M. Suozzo, and S. Sethumadhavan. 2013. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[29] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak. 2012. Hardware Trojan Horse Benchmark via Optimal Creation and Placement of Malicious Circuitry. In *Proc. of the ACM/IEEE Annual Design Automation Conference (DAC)*.

[30] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. 2016. Hardware Trojans: Lessons Learned After One Decade of Research. *ACM Transactions on Design Automation of Electronic Systems* (2016).

[31] J. Zhang and Q. Xu. 2013. On Hardware Trojan Design and Implementation at Register-Transfer Level. In *Proc. of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*.

[32] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu. 2013. VeriTrust: Verification for Hardware Trust. In *Proc. of the ACM/IEEE Annual Design Automation Conference (DAC)*.

[33] J. Zhang, F. Yuan, and Q. Xu. 2014. DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans. In *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Received April 2017; revised May 2017; accepted June 2017