Automatic Security Property Generation for Detecting Information-leaking Hardware Trojans

Chenguang Wang, Yici Cai and Qiang Zhou
Tsinghua National Laboratory for Information Science & Technology
Department of Computer Science & Technology, Tsinghua University, Beijing, China
Email: wang-cg13@mails.tsinghua.edu.cn, {caiyc, zhouqiang}@mail.tsinghua.edu.cn

***Abstract*—In recent years, formal methods have been adopted to detect the hardware Trojans (HT). However, they generally suffer from the time-consuming and error-prone development for property, lack of self-learning system to counter with the future HT types, and high computational complexity due to the growth of design scales. To overcome the above limitations, we propose an automatic security property generation method (ASPG) by feature analysis and property matching techniques. Machine learning is applied to systematically training the property library from the suspicious behaviors in unknown designs, which is expected to counter with the future HT. To reduce the computational complexity, we transform the register-transfer level (RTL) code into an introduced succinct abstract format to remove the redundant information which is unnecessary for depicting HT features. Experimental results show that the properties are generated in less than 50 ms with low memory consumption and the benchmarks from Trust-hub and DeTrust can be successfully detected with 0 false negatives and positives.**

I. Introduction

Recently, hardware Trojans (HT) has been one of the major concerns of the System-on-Chip (SoC) designers, which is commonly defined as the malicious modification inserted into the third-party intellectual property (3PIP) cores [1]. HT can cause various malicious effects, e.g. change of functionality, reliability reduction, and information leakage [2].

To counter with the increasing threat of HT, researchers have proposed various detection methods. Formal verification is to mathematically investigate the space of possible behaviors to check if a design satisfies the specification. Formal verification has been applied to detecting the HT, e.g. equivalence checking, theorem proving, and model checking.

However, the formal methods also have several limitations. 1) Iterative and tedious manual processing is required to develop the security properties used to check if a hardware design is HT-free, which is time-consuming and error-prone. The root reason is that there is a lack of automatic security property generation. 2) There is a lack of self-learning framework for the future and unknown HT types. As a result, the adversaries can adjust their design tactics with the existing detection techniques and thus the existing techniques may fail in detecting the future HT types. 3) Another limitation is the high computational complexity of property generation and verification due to the growing design scales.

This project is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61774091.

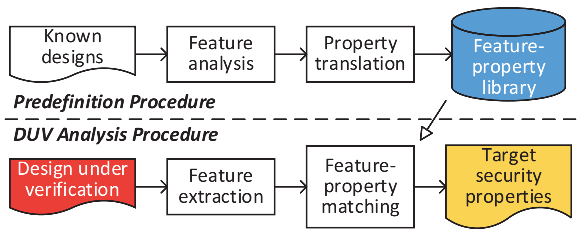


Fig. 1: Basic concept of the proposed ASPG.

To solve the problems, we propose an Automatic Security Property Generation framework for HT detection, i.e. **ASPG**. The basic concept of ASPG is to construct a feature-property library, and then match the features extracted from the design under verification (DUV) to the predefined library, generating the target security properties, as illustrated in Fig.1.

ASPG differs from the existing literatures in the following aspects. 1) To the best of our knowledge, ASPG makes the first attempt to automatically generate the security properties to alleviate the manual burdens in developing properties. 2) Machine learning technique, i.e. rule learning [3], is applied to systematically training the property lib, thus ASPG has the unique advantage of detecting the future HT types by the self-learning capability. 3) We propose a succinct abstract format for depicting the HT features in register-transfer level (RTL) netlist, i.e. coarse-grained control and data flow graph (CDFG), to remove the redundant information in CDFG, which significantly reduces the time and memory consumption.

To validate our method, security properties are generated and used to model check the HT-inserted and HT-free benchmarks from both Trust-hub [4] and DeTrust [5]. Experimental results show that the security properties can be generated within 50 ms with low memory consumption and the benchmarks can be detected with 0 false negatives and positives.

To summarize, the contributions of this paper are as follows:

1) To alleviate the manual burdens in developing properties for HT detection, we propose the methodology of ASPG to automatically generate the security properties by feature analysis and property matching techniques.
2) Machine learning algorithm, i.e. the rule learning, is applied in ASPG for systematically learning new properties by training the feature-property library to counter with the future and unknown HT types.

3) A succinct abstract format for depicting HT features, i.e. the coarse-grained CDFG, is introduced to remove the redundant information in traditional CDFG for reducing the computational complexity in property generation.

The remainder of this paper is organized as follows. Section II provides the background and preliminary. The proposed ASPG methodology is introduced in Section III. The experimental results are presented in Section IV. The conclusion and discussion are summarized in Section V.

II. Background and Preliminary

A. Formal Verification based HT Detection

There exit several methods based on equivalence checking and theorem proving. Fern *et al.* formulate the HT detection as a satisfiability problem using advanced Boolean and satisfiability modulo theory (SMT) solvers [6]. Farahmandi *et al.* propose to extract polynomials from gate-level implementation of the untrustworthy 3PIP and compare them with specification polynomials [7]. Based on theorem proving, Love *et al.* present a framework for the acquisition of provably trustworthy IP, which draws upon the research in the field of proof-carrying code (PCC) [8]. Further, Jin *et al.* introduce a proof-carrying based framework for asserting the trust of 3PIP, particularly the microprocessor cores [9]. Guo *et al.* propose a framework to combine an automated model checker with an interactive theorem prover for proving the security properties [10].

On the other hand, model checking has the significant advantage of golden reference-free checking and scalability for large designs. Zhang *et al.* present a comprehensive method with property checking, automatic test pattern generation (ATPG) and equivalence theorems to detect HT [11]. In [12], Rathmair *et al.* propose to apply various formal methods including model checking in a combinational manner to increase the introduced "Trojan Assurance Level", which is a metric for assessing the HT inexistence. Rajendran *et al.* develop the properties for detecting HT which corrupt critical data and verify the target design for satisfaction of these properties using a bounded model checker [13]. Subsequently, Rajendran *et al.* develop the properties to detect information-leaking HT [14]. In [15], Ngo *et al.* propose a synthesizable assertions module, i.e. the Hardware Property Checker (HPC), which verifies the permitted and prohibited behaviors of IC at run-time. Further, Guo *et al.* propose an integrated formal verification framework combining model checker with theorem prover to prove security properties [10] as mentioned above.

B. Model Checking and Automatic Property Generation

Model checking [16] is to mathematically check if a design satisfies a set of properties stated in temporal logic. In general, the design is modeled as a finite transition representation ($\mathcal{M}$) and model checking is to check if the property set ($\mathcal{P}$) holds on the model ($\mathcal{M} \models \mathcal{P}$). The properties can be expressed in Linear Tree Logic (LTL), SystemVerilog Assertion (SVA), and Property Specification Language (PSL)[1], etc.

[1]In this paper, our proposed method outputs the assertions in PSL format, while it is also able to output the LTL and SVA formats.

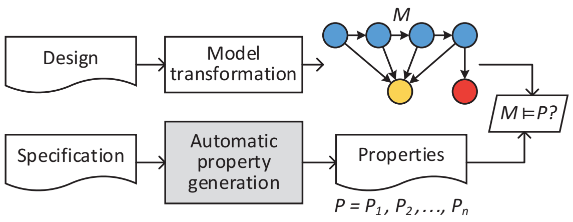


Fig. 2: Model checking with automatic property generation.

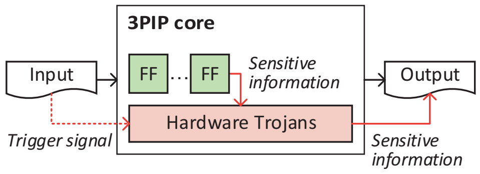


Fig. 3: Model of the information-leaking hardware Trojans.

In recent years, researchers have proposed several techniques to automatically generate the properties as well as assertions, as presented in Fig.2. Vasudevan *et al.* propose a decision tree based method to mine assertions from simulation traces [17]. In [18], Danese *et al.* propose a tool for automatic extraction of temporal assertions from execution traces of behavioral models by adopting a mix of static and dynamic techniques. Harris *et al.* present a method to automatically translate the specifications into SVA [19]. However, these techniques are introduced to cope with functional verification and thus not easy to be applied in the HT detection domain when HT results in malicious effects other than change of functionality, e.g. information leakage.

C. Threat Model

The threat model used in this paper follows the commonly used hardware threat model [13]. A rogue designer in 3PIP vendor is the attacker, whose purpose is to insert HT into the 3PIP cores. The 3PIP consumer is the defender, whose objective is to detect the HT (if any) in the 3PIP cores. We assume that the 3PIP cores under detection are delivered from the attacker to defender as the RTL netlist in VHDL/Verilog HDL formats without trustworthiness. Besides, we assume that both the detection procedure conducted by the defender and the used commercial tools are trustworthy.

In this paper, we focus on the *information-leaking* HT as the first attempt to the automatic security property generation, while the other attributes of HT, e.g. the trigger mechanism and location, are unconstrained. The information-leaking HT results in the malicious modification which leaks sensitive information, e.g. the encryption key of Advanced Encryption Standard (AES) engine, to the attacker through output data, as shown in Fig.3. It should be noted that the information can also be leaked through a covert side-channel [20]. The flip-flop (FF) is where the sensitive information is stored, and the red arrows represent the HT-infected data flows which are used to leak the information. As we can see in Fig.3, some HT types may have the absence of trigger signal, e.g. always-on HT.

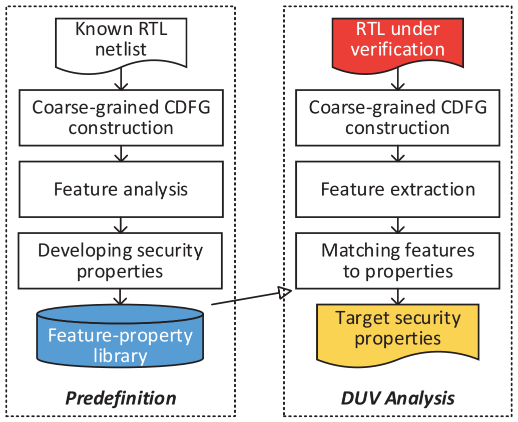

Known RTL netlist
Coarse-grained CDFG construction
Feature analysis
Developing security properties
Feature-property library
Predefinition
RTL under verification
Coarse-grained CDFG construction
Feature extraction
Matching features to properties
Target security properties
DUV Analysis

Fig. 4: Flow chart of ASPG.

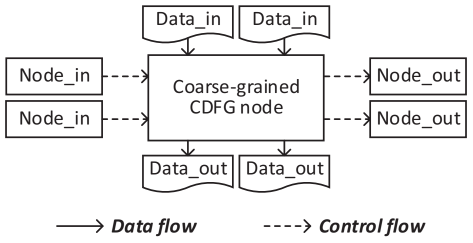
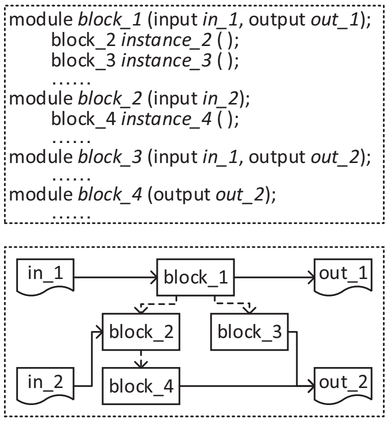


Fig. 5: A representative coarse-grained CDFG node.



```
module block_1 (input in_1, output out_1);
        block_2 instance_2 ( );
        block_3 instance_3 ( );
        ......
module block_2 (input in_2);
        block_4 instance_4 ( );
        ......
module block_3 (input in_1, output out_2);
        ......
module block_4 (output out_2);
        ......
```

Fig. 6: Sample RTL netlist in Verilog HDL format and the corresponding coarse-grained CDFG.

III. Methodology of ASPG

In this section, the details of ASPG will be introduced. As illustrated in Fig.4, ASPG consists of two procedures, i.e. ***Predefinition*** and ***DUV Analysis***. First, a security property library with the corresponding features, i.e. the feature-property lib, is constructed by feature analysis on the known designs. Second, the features extracted from the DUV are matched to the predefined lib, generating the target security properties. It can be seen that the flows and principles in the two procedures are similar, however, the difference is that the flows in *Predefinition* are semi-automatic (executed once and for all), while the *DUV Analysis* is fully automatic.

In order to reduce the computation complexity, we propose a succinct abstract format to depict the HT features. Using the introduced format, we investigate and summarize the features of information-leaking HT. Based on the features, we develop the corresponding security properties.

A. Coarse-grained CDFG

In this subsection, we propose a succinct abstract format of RTL netlist, i.e. coarse-grained CDFG. The most significant advantage of coarse-grained CDFG over a traditional CDFG is that *the redundant information in traditional CDFG which is unnecessary for HT feature analysis is removed*, reducing the computational complexity.

In the standard integrated circuit (IC) design flow, CDFG is commonly used as the internal representation in the high-level synthesis to capture the control and data flow information of RTL netlist. However, some of the information in CDFG is unnecessary for depicting the HT features. For example, at the level of single statement, it is difficult to distinguish the HT-infected one from the original one because they share a similar set of operations, e.g. add and shift. On the contrary, it has been noted that a higher level CDFG is able to distinguish the HT-infected with the original ones [21].

Based on the above observation, we propose a coarse-grained CDFG at the levels of module and basic block. The basic blocks here include the procedures, `if`/`case` blocks, loop blocks, and the statements blocks between them. Compared with the proposed coarse-grained CDFG, the traditional CDFG can capture more detailed information. Besides, the coarse-grained CDFG consists of control flow graph (CFG) and data flow graph (DFG). Formally, the coarse-grained CDFG is a directed acyclic graph (G) which can be expressed as:

$$G = (V, E) \tag{1}$$

In (1), V is the set of nodes which can be either module nodes or basic block nodes, and $E \subseteq V * V$ is the set of edges representing the transfer of either data or control from one node to another.

Fig.5 shows a representative coarse-grained CDFG node, where V is represented by a rectangle (representing the module and basic blocks), and the data and control flows are represented as black and dashed arrows respectively. Based on the definition, a sample RTL netlist[2] and corresponding coarse-grained CDFG are presented in Fig.6.

B. Features of Information-leaking HT

In this subsection, we investigate and summarize the features of information-leaking HT at the abstract level of coarse-grained CDFG. On the observation that there exist differences between the coarse-grained CDFG of HT-inserted and HT-free RTL, we summarize the features which can be used to check

[2]The sample RTL here only consists of modules due to the space limits, however, the basic blocks are also involved in the coarse-grained CDFG. Besides, it can be noted that an arbitrary RTL netlist in VHDL/Verilog HDL format can be modeled as a coarse-grained CDFG.

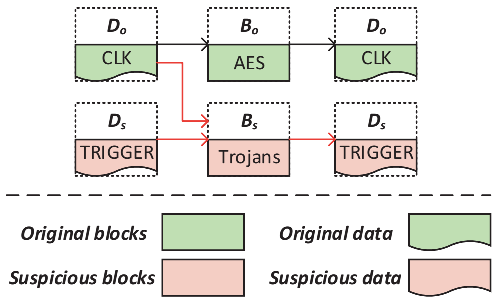


Fig. 7: Coarse-grained DFG of information-leaking HT.

if the DUV is HT-inserted. To simplify the investigation, we define the items as follows:

- **Original blocks** (B_o) are where the normal function is performed, e.g. the encryption block in AES.
- **Suspicious blocks** (B_s) are where the malicious effect of HT is performed, e.g. the key-leaking block in AES.
- **Original data** (D_o) is used to perform the normal function of B_o, e.g. the clock signal `CLK`.
- **Suspicious data** (D_s) is used to perform the malicious modification of B_s, e.g. the trigger signal `TRIGGER`.

Coarse-grained DFG: Fig.7 provides the coarse-grained DFG of information-leaking HT, where the red arrows indicate the HT-infected data flow of malicious modification and black arrows indicate the original data flow in the normal function. It can be noted that D_o can be used by both B_o and B_s as input or output because the system signals and sensitive data are also required to perform the malicious modification, e.g. the clock signal (`CLK`). On the contrary, D_s is only used by B_s because the HT-infected data, e.g. the trigger signal (`TRIGGER`), is not required to execute the normal function in B_o. Further, to hide the malicious operations, D_o is hardly altered for it is only read but not written by B_s.

Coarse-grained CFG: It can be noted that there exists no B_s which is called by B_o because the control flow is based on the data flow and there exists no D_s used by B_o as its input or output. Likewise, there must exist B_o which is called by B_s. To summarize, the normal function may be performed in both B_o and B_s, while the malicious modification is performed only in B_s.

Based on the above investigation in coarse grained CFG and DFG, the features of information-leaking HT can be formally expressed as (2a)-(2f) (referenced to as (2) for short), where I/O are respectively the set of input/output patterns, and D'/D'' are respectively the value of D (D_o or D_s) before/after being called by the blocks.

$$\exists D_o \in I \cup O \subset B_o \models D_o \in I \cup O \subset B_s \tag{2a}$$
$$\nexists D_s \in I \cup O \subset B_s \models D_s \in I \cup O \subset B_o \tag{2b}$$
$$\forall D_o \in O \subset B_s \models D''_o = D'_o \tag{2c}$$
$$\exists D_s \in I \subset B_s \models D''_s \neq D'_s \tag{2d}$$
$$\forall B_o \ \nexists B_s \models B_s \ is\ called\ by\ B_o \tag{2e}$$
$$\exists B_s \ \exists B_o \models B_o \ is\ called\ by\ B_s \tag{2f}$$

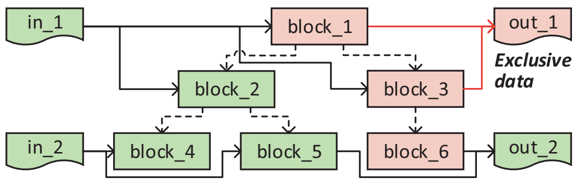


Fig. 8: Coarse-grained CDFG of always-on HT.

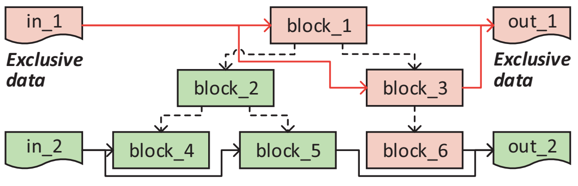


Fig. 9: Coarse-grained CDFG of immediate-triggered HT.

Equation (2) presents the theoretical foundations of ASPG. However, the problem here is that, how can we identify the B_o, B_s, D_o, and D_s in a coarse-grained CDFG?

To solve the above problem, we analyze the features in terms of the trigger mechanisms [1]. The first category is the *always-on* HT which is performed without a trigger. Second, the *immediate-triggered* HT are triggered when a predefined event in one clock cycle occurs, e.g. when a predefined input is detected. The *sequential-triggered* HT are triggered when a sequence of predefined events in multiple clock cycles occurs, e.g. when a predefined times of execution is detected. All the information-leaking HT can be classified into one of the three categories. The features are presented as follows.

- **Feature 1** (always-on HT): There exists a path of control flow which has exclusive data as output but not input, and the blocks in the path are B_s and the exclusive data is D_s, e.g. block_1, 3, 6 and out_1 in Fig.8.

We note the ***control path*** by the blocks from one node to the leaf node, and the ***exclusive data*** (input or output) for it is only used by the blocks in the path. It can be noted that there exists a path in the coarse-grained CFG which only consisting of B_s by (2e) and (2f). It is the same conclusion for the other HT types in this paper. Moreover, there exists no trigger but only payload in the always-on HT and D_s is used to implement the payload of HT according to (2d), which is noted as the exclusive output. There exists D_o which may be the input of both B_o and B_s by (2a). Based on above analysis, feature 1 is illustrated in coarse-grained CDFG as Fig.8.

- **Feature 2** (immediate-triggered HT): There exists a path of control flow which has exclusive data as input and output, and the blocks in the path are B_s and the exclusive data is D_s, e.g. block_1, 3, 6 and in_1, out_1 in Fig.9.

To the immediate-triggered HT, the payload is activated by a trigger signal which is enabled when the predefined event occurs. Consequently, there exists a trigger data which may be altered in B_s according to (2d), and this trigger signal is noted as the exclusive input data of the HT. The output of B_s is the same with Feature 1, as illustrated in Fig.9.

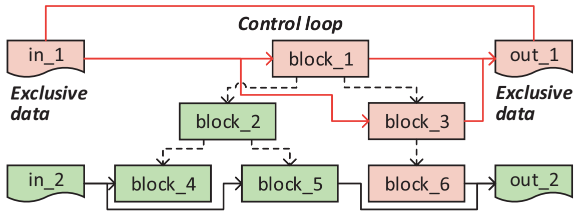


Fig. 10: Coarse-grained CDFG of sequential-triggered HT.

TABLE I: The features of information-leaking HT in coarse-grained CDFG format

#	Feature	Identification
1	A path in control flow has exclusive data as output	B_s: the blocks in the control path or loop B_o: the other blocks D_s: the exclusive data D_o: the other data
2	A path in control flow has exclusive data as input and output	
3	A loop in control flow has exclusive data as input and output	

- **Feature 3** (sequential-triggered HT): There exists a loop control which has exclusive data as input and output, and the blocks in the path are B_s and the exclusive data is D_s, e.g. block_1, 3, 6 and in_1, out_1 in Fig.10.

The most significant difference with the above features is the existence of ***control loop***, which is a loop of control and data flow. In the sequential-triggered HT, the payload is activated by a state signal which is enabled when the predefined sequence of events occurs. Consequently, the state signal is frequently altered by B_s. By (2c) and (2d), there exists D_s which is both input and output of B_s, producing a loop in the coarse-grained CDFG. This state signal is noted as the exclusive input data of HT, as shown in Fig.10.

Table I concludes the features of the information-leaking HT in terms of trigger mechanisms. Given a coarse-grained CDFG of unknown RTL, the B_o, B_s, D_o, and D_s can be identified based on the features.

C. Feature-property Library

Based on the summarized features, the corresponding security properties are presented in this subsection. As mentioned before, the feature-property library can be constructed with the security properties and the corresponding features.

- **Property 1** (corresponding to Feature 1): There is no trigger in always-on HT, and D_s generally stays changing, which is expressed as follows:

$$\nexists D_s \models D_s \in I \subset B_s \tag{3}$$
$$\forall D_s \in O \subset B_s \models f(D_s) \geqslant F_{upper} \tag{4}$$

In the definition, $f(D)$ is the transition frequency[3] of D, and F_{upper} is empirically set as the upper limit of frequency. As noted in Feature 1, there exists no trigger signal in the always-on HT, and the payload part (B_s) only has D_s as its output port. Moreover, it is noted that the always-on HT do not change their states frequently [11] because the HT is always performed during the circuit operation. As a result, D_s has a high-frequency transition activity. The above characteristics are respectively checked by (3) and (4), where the upper limit of transition frequency (F_{upper}) is empirically set.

- **Property 2** (corresponding to Feature 2): There is a trigger signal in immediate-triggered HT, and D_s generally stays stable, which is expressed as follows:

$$\exists D_s \models D_s \in I \subset B_s \tag{5}$$
$$\nexists D_s \in I \cup O \subset B_s \models (D_s \in I) \wedge (D_s \in O) \tag{6}$$
$$\forall D_s \in I \cup O \subset B_s \models f(D_s) \leqslant F_{lower} \tag{7}$$

As noted by Feature 2, there exists a trigger signal in the immediate-triggered HT which is used to activate the payload and checked by (5). Besides, there exists no D_s which is in I and O meanwhile, therefore there is no loop in the coarse-gained CDFG as shown in Fig.9, which is checked by (6). Compared with the always-on HT, D_s has a low-frequency transition activity for the HT are performed under rare conditions, as presented by (7).

- **Property 3** (corresponding to Feature 3): In sequential-triggered HT, there is a trigger signal and state data which flag the detected sequence of events, and D_s generally stays changing, which is expressed as follows:

$$\exists D_s \models D_s \in I \subset B_s \tag{8}$$
$$\exists D_s \in I \cup O \subset B_s \models (D_s \in I) \wedge (D_s \in O) \tag{9}$$
$$\exists D_s \in I \cap O \subset B_s \models f(D_s) \geqslant F_{upper} \tag{10}$$

In Property 3, the Feature 3 of sequential-triggered HT is checked. Similar with (5) in Property 2, the presence of trigger signal is checked by (8). Besides, there exists a state signal to flag the detected sequence of events which occur in multiple clock cycles, and the value of the state signal is altered frequently. As a result, there is D_s in I and O meanwhile, which is checked by (9). On the other hand, except the state signal, the other signals in D_s all have low transition frequencies, which is different with the case in Property 1. This characteristic is checked by (10).

Translation of property to the assertion: On the basis of the formally defined security property library, assertions can be easily and automatically generated, which are the inputs to a model checker with the RTL netlist to verify if the design contains HT. The translation of properties to PSL is performed with regards to the associated properties and the RTL netlist. For example, the signal TRIGGER is identified as D_s by Feature 2, then Property 2 is translated to the corresponding assertions. The PSL assertion derived from (7) is presented as follows, which checks if the signal TRIGGER has a transition frequency lower than 0.0001.

```
property P_LOW_FREQUENCY_TRANSITION_ACTIVITY =
always aes_start -> next_a[1:10000] (!TRIGGER);
assert P_LOW_FREQUENCY_TRANSITION_ACTIVITY;
```

[3]Rather than calculating the transition frequency, we develop the PSL properties using temporal operators, e.g. next_a, which describes the timing relationship between signals, expressions with a cycle interval.

D. Refinement with Rule Learning

As mentioned above, the adversaries can adjust their design tactics with the existing HT detection techniques, and thus the detection techniques may fail in detecting the future HT. To tackle this problem, we propose a refinement based on machine learning, i.e. the rule learning algorithms [3], to learn new properties from the unknown designs.

Given a set of training examples, rule learning aims at finding a set of rules which can be used for prediction or classification of new instances. Consequently, it is expected to counter with the threat of future HT types. To apply this technique into ASPG, it is crucial to select the proper characteristics on which the learning will be performed, i.e. the construction of training set. Generally, the selected characteristics should satisfy the following requirements. 1) They should be easy to extract. 2) They should be able to capture the necessary information of DUV. 3) They should be abstracted at the proper level so that the relationship between them and the behavior is easy to learn.

Considering the requirements, we select the features of coarse-grained CDFG to construct the training set. We have two classes of training examples, i.e. positive examples which are HT-inserted and negative examples which are HT-free. Our goal is to find a set of rules, composed of features processed from the previous procedures, which can be used to predict whether a new design contains HT. The notions are presented in Table II. We summarize the various characteristics of the coarse-gained CDFG, then 1 (*YES*) and 0 (*NO*) are used as the attribute values of characteristics of the training examples and testing instances. Besides, the examples are labeled with $\mathcal{C}$, i.e. HT-inserted or HT-free. In this way, the feature analysis is modeled as a rule learning problem.

Fig.11 provides an example of rule learning application, where the training set is constructed with 10 information-leaking DUVs from Trust-hub [4], including 5 HT-inserted designs and 5 HT-free designs. To a sequential-triggered Trojan, the existence of D_s which is in I and O meanwhile is noted as *D_S_EXIST_I_O*, and $e \in \mathcal{E}$, $i \in \mathcal{I}$ respectively have an attribute value of either 1 or 0 according to its corresponding coarse-grained CDFG. Given this training set and class attribute to the rule learning algorithms, `RULE1` can be learned because it covers all the HT-inserted examples, as expressed in (11).

$$D\_S\_EXIST\_I\_O == 1 \Rightarrow DUV\ is\ HT{-}inserted \quad (11)$$

It can be noted that the advantage of the refinement with rule learning is that a complete and systematic set of security properties can be learned, and our method is able to detect the future and unknown HT types by self-learning.

Overfitting problem: Even through this refinement has proved effective in our method, overfitting is a well-known problem in machine learning, and the reason is either too many characteristics or too few training examples. We propose to learn from smaller groups of characteristics, based on the observation that not all the characteristics are independent.

TABLE II: The notions of rule learning application

Symbol	Notion	Meaning in Our Application
$\mathcal{E}$	Training set	Known designs with $\mathcal{C}$
$\mathcal{I}$	Testing set	DUV without $\mathcal{C}$
$\mathcal{C}$	Classification	HT-inserted or HT-free
$\mathcal{R}$	Set of rules	Library of security properties
$\mathcal{H}$	Hypothesis	Induction of $\mathcal{R}$
$Cov(r, e)$	Coverage function	Define if $r \in \mathcal{R}$ covers $e \in \mathcal{E}$

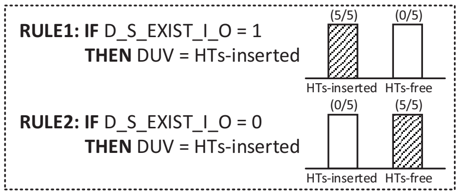


Fig. 11: Example of the rule learning application.

For each 100%-accurate rule, i.e. $Cov(r, \mathcal{E}) = 100\%$, a group is constructed containing all the characteristics used by this rule. In this way, the characteristics are partitioned into smaller groups. Then, rule learning is performed for each group, and the rule with the highest accuracy is selected as the final learned rule.

IV. Experimental Results

To validate the effectiveness and efficiency of the proposed method, we check the HT-inserted and HT-free benchmarks from Trust-hub [4] using the generated security properties with the model checker, Cadence Incisive Formal Verifier (IFV). We also check the stealthier implicitly-triggered HT-inserted benchmarks from DeTrust [5], which have successfully defeated several HT detection methods. To verify the effectiveness of rule learning applications, we perform a comparison with the manually summarized properties, which will be presented in this section. Fig.12 presents the overall experimental framework. ASPG is implemented in C++ language and executed on a Linux server with Intel Xeon E5604 CPU @ 2.67 GHz and 64GB RAM.

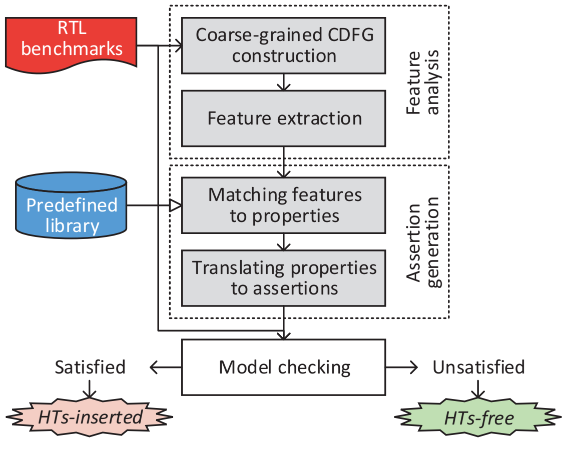


Fig. 12: Flow chart of experimental framework.

TABLE III: Detection capability of ASPG on the HT-inserted benchmarks from both Trust-hub [4] and DeTrust [5]. The "Candidate" indicates the number of B_s or D_s which are identified by ASPG, and the "Hit" indicates the number of B_s or D_s in "Candidate" which are actually HT-infected.

Benchmark	**Trigger Type**	**Suspicious Blocks**		**Suspicious Data**		**Detected?**	**Peak Memory (MB)**	**Time (ms)**		
		Candidate	**Hit**	**Candidate**	**Hit**			**Feature Analysis**	**Assertion Generation**	**Total**
`AES-T100`	Always-on (Feature 1)	**14**	12	9	9	Yes	11.58	13.871	8.120	21.991
`AES-T200`		13	13	**10**	9	Yes	13.03	13.262	8.662	21.924
`AES-T300`		**14**	13	**11**	9	Yes	13.02	13.149	7.991	21.140
`AES-T400`	Immediate-triggered (Feature 2)	**18**	17	9	9	Yes	13.03	13.786	9.299	23.085
`AES-T600`		12	12	8	8	Yes	13.03	13.221	10.481	23.702
`AES-T700`		14	14	10	10	Yes	12.63	13.330	11.624	24.954
`AES-T1300`		11	11	**9**	8	Yes	12.65	13.664	10.218	23.882
`AES-T2000`		11	11	8	8	Yes	12.63	13.264	9.653	22.917
`BasicRSA-T100`		11	11	9	9	Yes	9.47	3.709	5.222	8.931
`AES-T800`	Sequential-triggered (Feature 3)	14	14	10	10	Yes	13.05	13.690	18.808	32.498
`AES-T900`		14	14	10	10	Yes	12.62	13.187	19.019	32.206
`AES-T1600`		17	17	9	9	Yes	13.03	13.437	18.677	32.114
`BasicRSA-T300`		12	12	9	9	Yes	9.65	3.686	5.578	9.264
`PIC16F84`		12	12	8	8	Yes	9.32	2.285	2.325	4.610
`Or1200_ctrl`		11	11	8	8	Yes	9.79	2.061	2.146	4.207

TABLE IV: Comparisons with the existing works. For the limitation of reported results, `AES-T700` is selected as the reference benchmark because it is the most commonly used in the works. "N/A" indicates that no reported result is available.

Works	**Manual Processing during Verification**	`AES-T700`		
		Detected?	**Memory (MB)**	**Time (ms)**
Guo *et al.* [10]	Semantic translation of HDL code and properties to *Gallina* language	N/A	N/A	N/A
Zhang *et al.* [11]	Translation of the specifications to properties and assertions	N/A	N/A	N/A
Rajendran *et al.* [13]	Generation of the security properties and corresponding assertions	Yes	2058.24	40600
Rajendran *et al.* [14]	Generation of the security properties and corresponding assertions	Yes	4043.56	96330
Yao *et al.* [21]	Examination of the candidates that match HT features	Yes	757.2	2855.553
This paper	**None** (the property lib is predefined once and for all)	Yes	**12.63**	**24.954**

A. Detection Capability

Table III shows the results on HT-inserted benchmarks. As we can see, the HT can be detected with 0 false negatives. The same experiments on the HT-free ones show 0 false positives in HT detection, which is not presented due to the space limits.

It can be seen that 3 genuine blocks have been identified as suspicious (in bold). This can be contributed to the adder and multiplier statements in RTL, which produces the relationship between B_o and B_s. As a result, B_o can be identified as B_s. Likewise, D_o can be identified as D_s. Moreover, the false identification rates decrease with the presence of trigger signal in immediate-triggered HT and loop control in sequential-triggered HT, which makes it easier to identify B_o/B_s and D_o/D_s. The last three columns show the consumed memory and time of ASPG, which is within reasonable bounds.

To summarize, ASPG is able to generate the target security properties in less than 50 ms with low memory consumption, and the HT-inserted and HT-free benchmarks can be successfully detected with 0 false negatives and positives.

B. Comparison with Existing Works

Our proposed method is based on the theoretical foundations of automatic security property generation, which is different with the existing literatures. Moreover, a unified metric for HT detection capability is lacking, thus, a fair and elaborate comparison is not currently possible. Based on the reported results on the most frequently used benchmark, i.e. `AES-T700`, a rough comparison is concluded in Table IV.

The final results of the integrated formal verification framework in [10] have not been reported, and the results in [11] are presented using a proposed metric of "SS-Overlap-Trojan", i.e. the ratio of number of suspicious signals (SS) and Trojan signals. The consumed time and memory in [13] and [14] are fairly high. In [21], Yao *et al.* propose a method to analyze the HT features at the abstract level of flip-flop CDFG of the circuits. Obviously, the flip-flop level CDFG incurs a higher computational complexity than our technique because of the redundant information in CDFG.

Compared with the existing literatures, it can be noted that the manual work in developing property is significantly relieved. Moreover, our technique incurs a lower memory and time consumption.

C. Improvement by Rule Learning

We add the coarse-grained CDFG of benchmarks except for the `AES-Txxx` from Trust-hub into the training set (both HT-inserted and HT-free), while the `AES-Txxx` benchmarks are added into the testing set. The number of training samples is 337, and 27 rules are generated which are added into the feature-property library. On the observation that ASPG is able to check both the HT-inserted and HT-free benchmarks with 0 negative and positive false, we introduce a metric to assess the generalization for future and unknown HT types, i.e. "Hit-to-candidate ratio" (*HCR*), which defines the identification ability of suspicious blocks as *hits*/*candidates*.

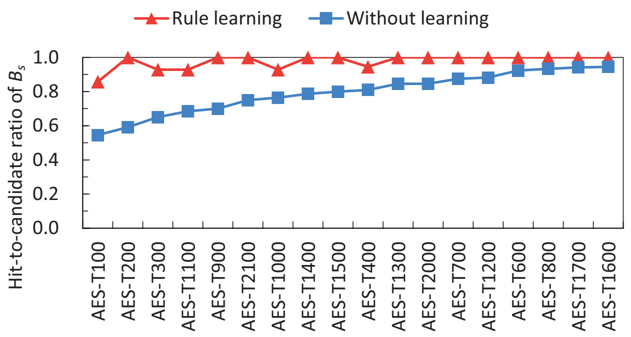


Fig. 13: Hit-to-candidate ratio improvement by rule learning.

Fig.13 presents the results of rule learning. It can be seen that 13 out of 18 benchmarks have 100%-*HCR* by the rule learning, while no benchmark has 100%-*HCR* by the method without learning. The results demonstrate that the rule learning algorithms can obtain an improvement on the identification ability of B_s which is crucial for the success of HT detection, thus, it is expected that ASPG can counter with the future and unknown HT types with the self-learning ability.

V. Conclusion and Discussion

In this paper, we explore the automatic property generation for HT detection. We perform a detailed investigation on the HT type of information-leaking at the introduced abstract level of coarse-grained CDFG and summarize the features. Further, based on the features, we develop the corresponding security properties and produce the feature-property library, which can be used to check if a design contains the information-leaking HT. In the *DUV Analysis* procedure, the extracted features from DUV are matched to the property library, generating the target security properties automatically. Experimental results on the benchmarks from Trust-hub and Detrust show that the properties are generated in less than 50 ms with low memory consumption, and the HT-inserted and HT-free RTL can be successfully detected with 0 false negatives and positives.

The first point we should highlight is that the manual work is significantly relieved, which can be contributed to the semi-automatic *Predefinition* (executed once and for all) and fully automatic *DUV Analysis*. ASPG has the unique advantage of self-learning ability. As the adversaries can adjust their design tactics with the existing HT detections, our technique is expected to cope with the future HT types. Besides, ASPG has a lower consumption of memory and time which can be contributed to the introduced succinct and higher abstract level of coarse-grained CDFG, with the redundant information removed for reducing the computational complexity.

In the future, we plan to propose a cross-level CDFG. The features of HT can be extracted at various levels, and the high-level has the advantage of low computational complexity while the low-level has a stronger ability to capture more detailed information. It is a trade-off between efforts and detection capacity. A cross-level CDFG is a potential to improve the detection capacity for future and stealthier HT by combining the high-level and low-level CDFG. Another important issue for future research is the overfitting problem due to the limited number of the training set. Based on the characteristic-grouping technique, we plan to partition the characteristics into several smaller groups by combining the relevant ones to reduce the total number of characteristics in the training set.

References

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.
[2] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2012.
[3] J. Fürnkranz, D. Gamberger, and N. Lavrac, *Foundations of Rule Learning*, ser. Cognitive Technologies. Springer, 2012.
[4] Trust-hub. http://www.trust-hub.org.
[5] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014, pp. 153–166.
[6] N. Fern, I. San, and K. T. Cheng, "Detecting hardware trojans in unspecified functionality through solving satisfiability problems," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC*, 2017, pp. 598–504.
[7] F. Farahmandi, Y. Huang, and P. Mishra, "Trojan localization using symbolic algebra," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC*, 2017, pp. 591–597.
[8] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.
[9] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, 2013, pp. 824–829.
[10] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *IEEE International Symposium on Hardware Oriented Security and Trust, HOST*, 2016, pp. 124–129.
[11] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital IP cores," in *IEEE International Symposium on Hardware-Oriented Security and Trust, HOST*, 2011, pp. 67–70.
[12] M. Rathmair, F. Schupfer, and C. Krieg, "Applied formal methods for hardware trojan detection," in *IEEE International Symposium on Circuits and Systemss, ISCAS*, 2014, pp. 169–172.
[13] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Design Automation Conference, DAC*, 2015, pp. 112:1–112:6.
[14] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *International Conference on VLSI Design, VLSID*, 2016, pp. 547–552.
[15] X. T. Ngo, J. Danger, S. Guilley, Z. Najm, and O. Emery, "Hardware property checker for run-time hardware trojan detection," in *European Conference on Circuit Theory and Design, ECCTD*, 2015, pp. 1–4.
[16] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2001.
[17] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, W. Tuohy, and D. R. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation and Test in Europe, DATE*, 2010, pp. 626–629.
[18] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Design, Automation and Test in Europe, DATE*, 2015, pp. 67–72.
[19] C. B. Harris and I. G. Harris, "Glast: Learning formal grammars to translate natural language specifications into hardware assertions," in *Design, Automation and Test in Europe, DATE*, 2016, pp. 966–971.
[20] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering," in *Cryptographic Hardware and Embedded Systems - CHES*, 2009, pp. 382–395.
[21] S. Yao, X. Chen, J. Zhang, Q. Liu, J. Wang, Q. Xu, Y. Wang, and H. Yang, "Fastrust: Feature analysis for third-party IP trust verification," in *IEEE International Test Conference, ITC*, 2015, pp. 1–10.